Croquet Programming

A Concise Guide

Draft 0.14

David A. Smith

Andreas Raab
David P. Reed
Alan Kay

Other Croquet Documents:
Croquet User Guide (to be written)
Croquet System Architecture (to be written)
Croquet Reference Guide (to be written)
Croquet Cookbook (to be written)
Croquet Programming (this document)
Programming OpenGL Using Croquet (to be written)

460 S. California Ave. #304
Palo Alto, CA 94306
www.qwaq.com

Table of Contents

Introducing Croquet

Croquet was built to answer a simple question. “If we were to create a new operating system and user interface knowing what we know today, how far could we go?” Further, what kinds of decisions would we make that we might have been unable to even consider 20 or 30 years ago, when the current operating systems were first created?

The landscape of possibilities has evolved tremendously over the last few years. Without a doubt, we can consider Moore’s law and the Internet as the two primary forces that are colliding like tectonic plates to create an enormous mountain range of possibilities. Since every existing OS was created when the world around it was still quite flat, they were not designed to truly take advantage of the heights that we are now able to scale.

What is perhaps most remarkable about this particular question is that in answering it, we find that we are revisiting much of the work that was done in the early sixties and seventies that ultimately led to the current successful computer architectures. One could say that in reality, this question was asked long ago, and the strength of the answer has successfully carried us for a quarter century. On the other hand, the current environments are really just the thin veneer over what even long ago were seriously outmoded approaches to development and design. Most of the really good fundamental ideas that people had were “left on the cutting room floor”.

That isn’t to say that the early pioneers thought of everything either. A great deal has happened in the last few decades that allows for some fundamentally new approaches that could not have been considered at the time.

We are making a number of assumptions:

- Hardware is fast – really fast, but other than for booting Windows or playing Quake no one cares – nor can they really use it. We want to take advantage of this power curve to enable a richer experience.

- 3D Graphics hardware is really, really fast and getting much faster. This is great for games, but we would like to unlock the potential of this technology to enhance the entire user experience.

- Late bound languages have experienced a renaissance in both functionality and performance. Extreme late-bound systems like LISP and Smalltalk have often been criticized as being too slow for many applications, especially those with stringent real-time demands. This is simply no longer the case, and as Croquet demonstrates, world-class performance is quite achievable on these platforms.

- Communication has become a central part of the computing experience, but it is still done through the narrowest of pipes, via email or letting someone know that they have just been converted into chunks in Quake. We want to create a true collaboration environment, where the computer is not just a world unto itself, but a meeting place for many people where ideas can be expressed, explored, and transferred.

- Code is just another media type, and should be just as portable between systems. Late binding and component architectures allow for a valuable encapsulation of behaviors that can be dynamically shared and exchanged.

- The system should act as a virtual machine on top of any platform. We are not creating just another application that runs on top of Windows, or the Macintosh – we are creating a Croquet Machine that is highly portable and happens to run bit-identical on Windows, Macintosh, Linux, and ultimately on its own hardware… anywhere we have a CPU and a graphics processor. Once the virtual machine has been ported, everything else follows; even the bugs are the same. Most attempts at true multiplatform systems have turned out to be dangerous approximations (cf. Java) rather than the bit-identical "mathematically guaranteed" ports that are required.

- There are no boundaries in the system. We are creating an environment where anything can be created; everything can be modified, all in the 3D world. There is no separate development environment, no user environment. It is all the same thing. We can even change and author the worlds in collaboration with others *inside them while they are operating*.

The existing operating systems are like the castles that were owned by their respective Lords in the Middle Ages. They were the centers of power, a way to control the population and threaten the competition. Sometimes, a particular Lord would become overpowering, and he would declare himself as King. This was great for the King. And not too bad for the rest of the nobles, but in the end – technology progressed and people started blowing holes in the sides of the castles. The castles were abandoned. Technology enables this.

This Document

This document is an introduction to developing new objects and applications for the Croquet collaboration platform. We describe the Croquet sdk 1.0 β release. Croquet is evolving rapidly, so you should expect this document to evolve as well. You should check for updates to both this document and the Croquet sdk release on a regular basis.

This Croquet sdk 1.0 β release is intended for somewhat more sophisticated programmers that already know some Smalltalk and can use the Squeak code browser editors. It also presumes that the user knows something about OpenGL, or is at least willing to learn in parallel to this document.

The Croquet User Manual is not an attempt to teach you how to use the Squeak programming environment, or how to program in OpenGL. To use the Croquet SDK 1.0, you will need to learn how to start and stop Squeak, manipulate objects, and examine and change code. There are a number of tutorials, books, and user groups to help. See http://squeak.org for more information and the book *Squeak: Object-Oriented Design with Multimedia Applications* by Mark Guzdial, Prentice-Hall, 2000.

There are a number of useful resources available for learning OpenGL, including the *OpenGL Programming Guide* by Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, Third Edition. Addison-Wesley. 1999 and OpenGL Game Programming by Kevin Hawkins and Dave Astle, Prima Publishing, 2001.

The Croquet Programming Guide (this document) is focused on the overall use and architecture of Croquet. Later versions of this document will include additional discussion about the development environment and graphics programming model, because these will begin to diverge somewhat from the traditional Squeak world. This is a living document, and we plan that it will continue to evolve just as we plan that Croquet will evolve. This document should become the first reference for Croquet users and developers. Please send comments, error reports, etc. to david.smith@qwaq.com . We really want to make Croquet and this document as useful and valuable as possible.

The following formatting conventions are used throughout this document:

Code is displayed as below:

```
"This is some code."
#turn: angle

        cube rotateAroundY: angle.
        (self future:100)turn: angle+1.
```

Any numbers to the left are used for referencing the lines of code in the text.

General notes are displayed this way:

> Notes – like this one, are used when I describe transient parts of the system – those things that will very likely change, or require a lot of additional effort to work with. The notes may even contain suggested modifications to the system and warnings to those brave enough to venture into developing anything interesting in Croquet.

Why Croquet?

Croquet enables cooperation at all levels. For example:

An application written using the Croquet SDK is automatically collaborative between people. Users can see each other and what they are doing in real-time. The result of their work is immediately available to other users. Neither the users nor the programmer needs to do anything special to make this happen.

Application objects in Croquet share a common protocol that allows them to cooperate with each other. For example, every Croquet object can be attached to another Croquet object so that it moves in a fixed relationship to that other object. A programmer uses the Croquet libraries to render the object individually, and the system will take care of rendering it in relationship to all the other objects. It can have sound and the system will take care of combining the sound of all the objects with respect to their relative positions to the user. This is true even for those application objects that are just display wrappers for something in another system external to Croquet. All this allows separately developed multimedia and other rich content to be combined by cooperating users into cooperating objects.

All of the source code is available to everyone and can be used for public, private, and commercial purposes. Programmers can share their code with others or build the community in other ways, or they can choose not to.

Squeak

Croquet runs within Squeak, a highly capable cross-platform open-source implementation of Smalltalk.

The Smalltalk language provides a very powerful, flexible, yet easy-to-learn language for writing Croquet applications and machinery. It is not difficult to examine Smalltalk code that was written by someone else. Croquet applications are built by defining application objects with specific behavior, and Smalltalk has traditionally been the language used for discussing programs in this way. This programming guide will not teach Smalltalk, but there are many tutorials, books, and user-groups available.

The Squeak implementation of Smalltalk provides a single environment in which applications are developed and used. There is not a separate "run-time" environment: instead, all programming tools are available at all times even while the application is running. Changes can be made dynamically, without needing to restart the application and recreating the application state.

Squeak consists primarily of four files:

- The source code for everything, which is the same for all platforms, and which is examined (and even changed) using Squeak itself.
- A small, fast, virtual machine that executes compiled Squeak code. There is a different virtual machine executable for each type of computer that Squeak runs on.
- A pair of files, called the .image file and the .changes file, which operate together to provide the complete state of Squeak and all object definitions. Because Squeak object memory is exactly the same on every platform, these two files are identical across all platforms. These files contain all the Croquet object definitions.

In addition, Croquet comes with some extra machine-specific libraries, called plug-ins, that extend the Squeak virtual machine.

There are a variety of application windowing systems available in Squeak. For this version of the Croquet SDK, the top level Croquet object (and the demo variations) use the Morphic window system, and run as a single Morphic window within Squeak.

Do NOT depend upon Morphic in your development process in any way. Morphic will be replaced by Tweak in the Croquet SDK 1.1 release. Further, Morphic was never designed to be a collaborative framework where Tweak shares Croquet's low-level architecture.

Using Croquet

Getting Started

Starting Squeak

Croquet is provided with an .image file that contains all the original Croquet object definitions. In this release, it is named *Croquet1.0.n.image, where n is an integer*. However, you may be using a newer file with additional or changed definitions, created by you or someone else. You will need to know the name of this .image file.

Squeak can be started by dragging and dropping the .image file onto the virtual machine executable for your platform. There are platform-specific variations:

Windows: The virtual machine executable is named *Croquet exe*. If you just double-click on the .exe without dragging anything on to it, the .exe will automatically find an .image file to use. If there is only one .image file, it will use that without asking. If there are multiple .image files, you will be prompted to select one. You can also arrange for Windows to use the .exe whenever you double-click on the .image file. The way to do this is to double-click on the .image file, browse for the .exe, and click "Always use the selected program to open this kind of file."

Macintosh: The virtual machine executable is named *Croquet.app*. If you just double-click on the .app without dragging anything on to it, the .app will prompt you for an .image file. You can also arrange for MacOS to use the .app whenever you double-click on the .image file. The way to do this is to ctrl-click on the .image and choose Open With -> Other. Then select the .app and click the "Always Open With" box. Since the default settings on Mac OS do not display the ".app" extension you will may see an application called something like `Croquet` with an icon like below.

Linux: The virtual machine executable is named *croquet*. Linux installations vary, but executing *INSTALL* in the linux installation directory should set things up for most systems, including the placement of the virtual machine executable in a place like */usr/local/bin*. With the location of the virtual machine executable in your path, you should be able to type *croquet croquet1.0.n* on the command line to start

Squeak (where n is replaced by an integer corresponding to the image you want to use).

Before starting, make sure that OpenGL and OpenAL libraries are installed and available on your computer. Croquet should look for the libraries in by searching the LD_LIBRARY_PATH, but depending on how you have arranged you system, you may need to make symlinks or move the libraries. The libraries are:

libGL.so
libopenal.so

If you are missing these libraries see:

http://www.opengl.org/
http://www.openal.org/

to get them. We know that croquet works under Fedora Core 4.0 linux distro with OpenGL 2.0 and OpenAL 0.0.8. Your mileage may vary.

Once you have the audio and graphics support libraries in place, unzip the croquet distribution. Suppose you name the directory holding this software "croquet-beta". In this example, to run Croquet:

```
cd croquet-beta

./croquet.sh
```

The croquet.sh shell script makes sure that a couple important links are in place, then launches the croquet virtual machine with the croquet.image file found in your croquet directory. You might want to modify this. Here is the shell script source:

```
#!/bin/sh
DIR=`dirname $0`
EXE="$DIR/bin/i686-pc-linux-gnu"

# make source link if necessary

if [ ! -r $DIR/CroquetV1.sources ] ; then
      ln -s $DIR/bin/CroquetV1.sources $DIR/CroquetV1.sources
fi

# make libGL.so link if necessary
if [ ! -x /usr/lib/libGL.so -a -x /usr/lib/libGL.so.1 ] ; then
```

```
    if [ ! -x  "$EXE/libGL.so" ] ; then
        echo Creating libGL.so symlink in $EXE
        ln -sf /usr/lib/libGL.so.1 "$EXE/libGL.so"
    fi
fi

exec "$EXE/squeak" -plugins "$EXE" \
    -vm-display-X11 -swapbtn \
    "$DIR/croquet.image"
```

Stopping Squeak

Click on the Squeak desktop -- anywhere where there isn't something else. From the "World" menu that pops up, select "quit". (The last item.)

Starting Croquet

The Croquet .image starts with a button visible labeled "Start Croquet Demo (master)." Click on this button to run the Croquet demo worlds full-screen. If someone on your local area network is already running the Croquet demo, you can instead press the button labeled "Join Croquet Demo."

There are other Croquet "applications," available. The green object browser window shows these. You can click on one and drag it anywhere outside the green box. The application will run in a small window, allowing you to use other Squeak tools while Croquet is running.

Stopping Croquet

To quit Croquet, you delete the Squeak Morph that forms the top-level window for Croquet. This is done by bringing up the Morphic menu, which is platform-specific:

Windows: Alt-click on the Croquet window. (Hold down the Alt key and then press the left mouse button.)

Macintosh: Apple-click on the Croquet window. (Hold down the apple or "flower" key and then press the mouse button.)

Linux: Alt-click on the Croquet window. (Hold down the Alt key and then press the left mouse button.)

Morphic will then bring up a "halo" of buttons around the Croquet window. To quit Croquet, click the pink 'X' button in the upper left corner.

Default User Interface

The Croquet platform is designed to support any number of different user interfaces. With the majority of development effort going into core Croquet functionality, this release provides a single top-level user interface (although individual objects in the space can define their own interfaces). The following sections describe how to interact with Croquet via this interface; the reader should bear in mind that this interface is not a fundamental part of Croquet.

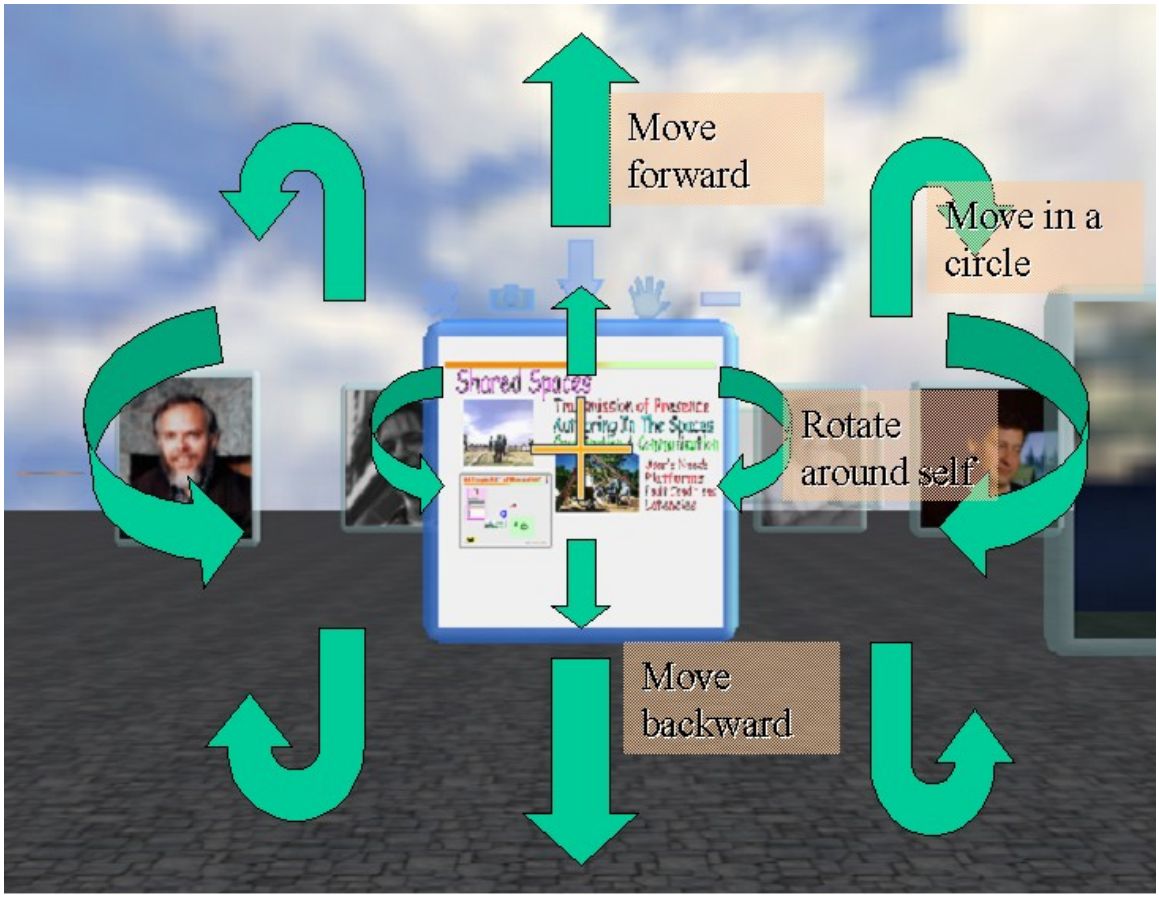
Only a keyboard and mouse are required to interact with Croquet. As is traditional in Smalltalk environments, Croquet supports three-button mice. The default interface makes uses of two, which we name the “manipulation” and “navigation” buttons. Keyboard modifiers enable the use of mice with only a single button.

Movement

The “navigation” button depends on the underlying operating system as follows:

> Windows: right-click (Press the right mouse button.)
>
> Macintosh: option-click. (Hold down the option key and then press the mouse button.) If you have a three-button mouse, you can also press the right mouse button.
>
> Linux: Same as Windows.

To drive your avatar around, press and hold the navigation mouse button within the Croquet window, at some distance from the center of the window. The closer your mouse is towards the top, the faster you move forward. The closer your mouse is towards the bottom, the faster you move backwards. Similarly for turning left or right. If you also hold down the shift key, you will look up and down rather than moving forwards or backwards, respectively.



You can move vertically in the space using the arrow keys. Note, however, that in most Croquet worlds, any object below you will apply "gravity," such that when you stop pressing, e.g., the up arrow button, you will gradually fall back down to the object. You will not fall through an object. You can also use the mouse's scroll-wheel to move up and down.

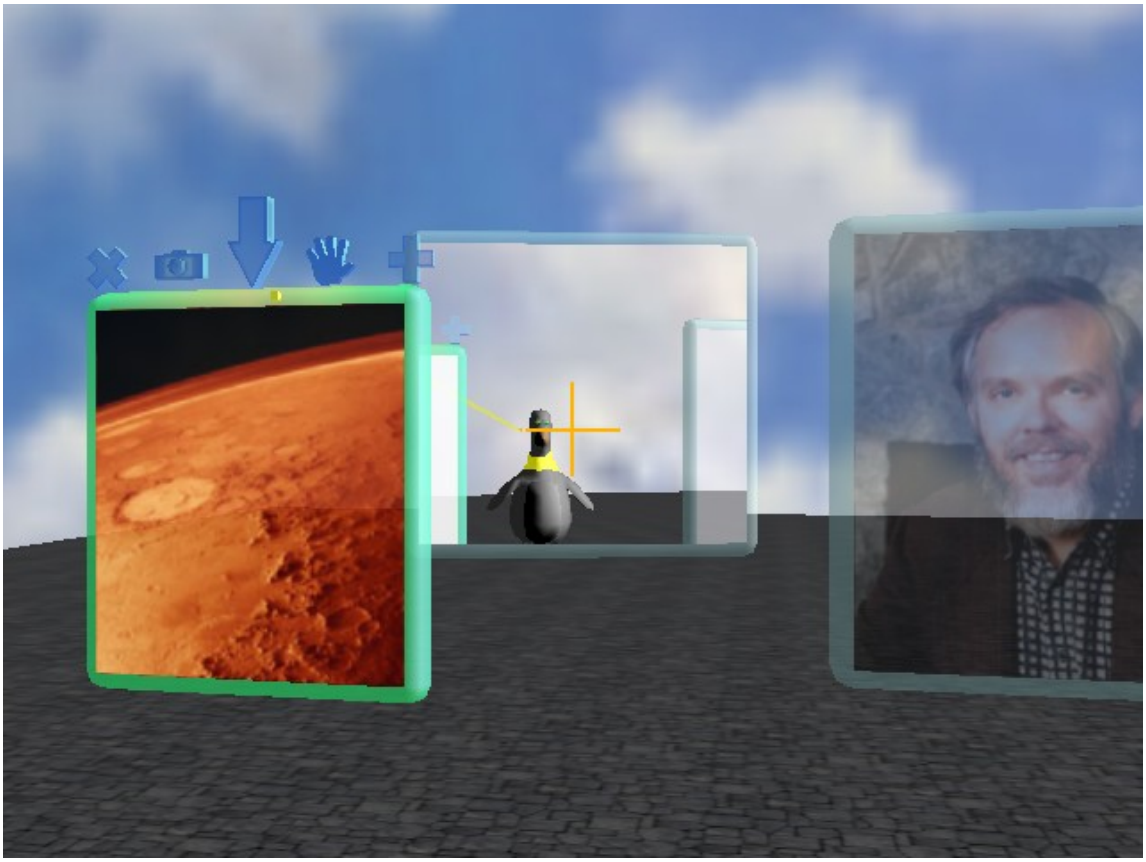
Picking/Manipulation

Picking and manipulating 3D objects in Croquet is a central feature of the system. It is what gives the users the ability to create and manipulate their environment and allows them to interact with it in a way that simply wasn't possible before. We describe three examples of picking and manipulating objects here.

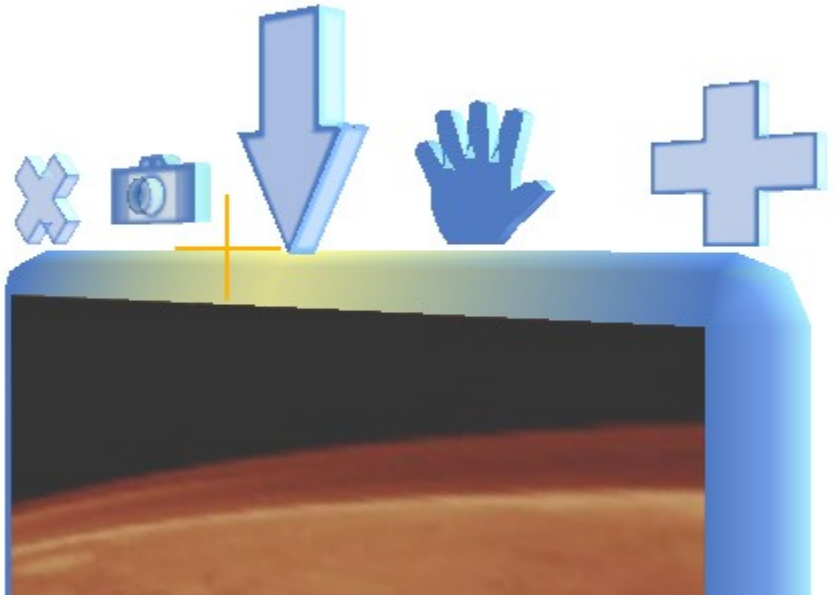
3D Windows

There are several flat rectangular window objects in Croquet that all define the same default behavior for manipulation:

Moving the mouse over the window causes buttons to appear above the top of the window. These will fade away in time if not used. The buttons are activated by clicking them.



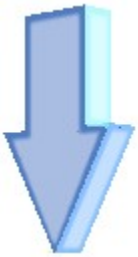
The window to the left has an active set of buttons along the top. The other windows displayed, (the mirror in the background for example) don't. If you move the cursor over these windows, the halo will instantly appear.



From left to right the buttons are:


The kill button removes the window and its contents if there is no other reference to them.

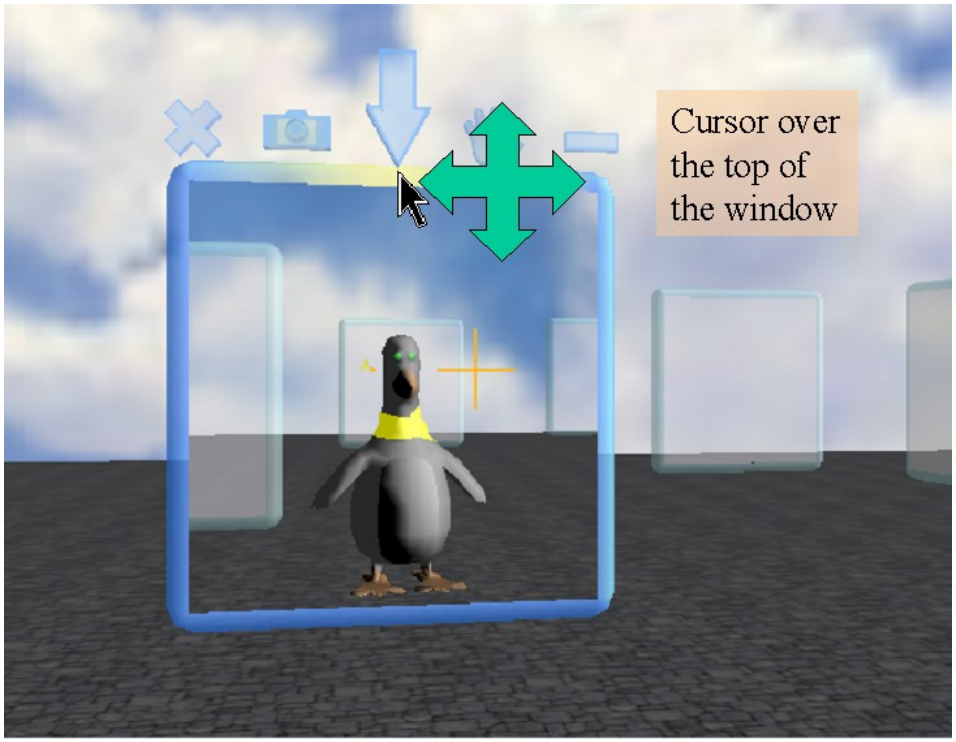

The camera button takes a picture of the contents of the window's contents, and (if the window contains a portal) records the current viewpoint as "landmark" that can be returned to later. There will be more on landmarks and portals later. Currently, this is undefined for a window that does not have portal contents.


This down arrow button aligns the viewpoint of the camera (the user's viewpoint) directly on the window such that it is flat and takes up the entire view screen. This is very useful for viewing and editing documents.

The hand button allows the user to grab the window. The window will now “follow” the user around no matter where he goes, until he selects the hand button again to release it. The window is then left where the camera and user left it.

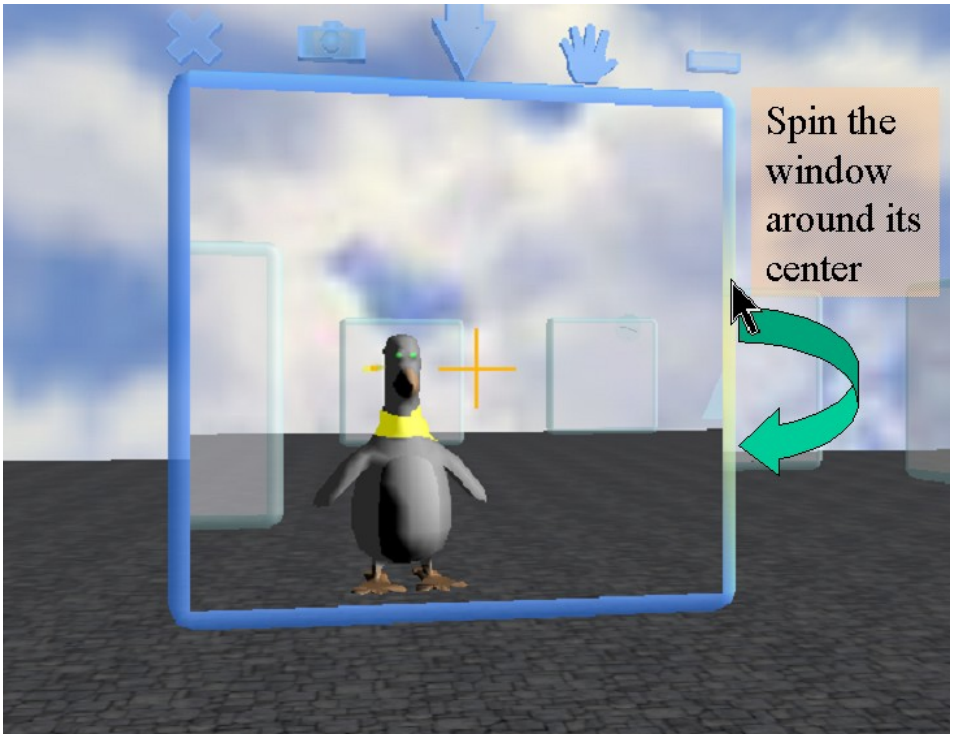
The plus button opens the contents of the window – which are currently closed. Once the window is opened, the contents are visible. This button icon then turns into a minus button, which allows the window to be closed again.

Several parts of the window frame are also active individually:

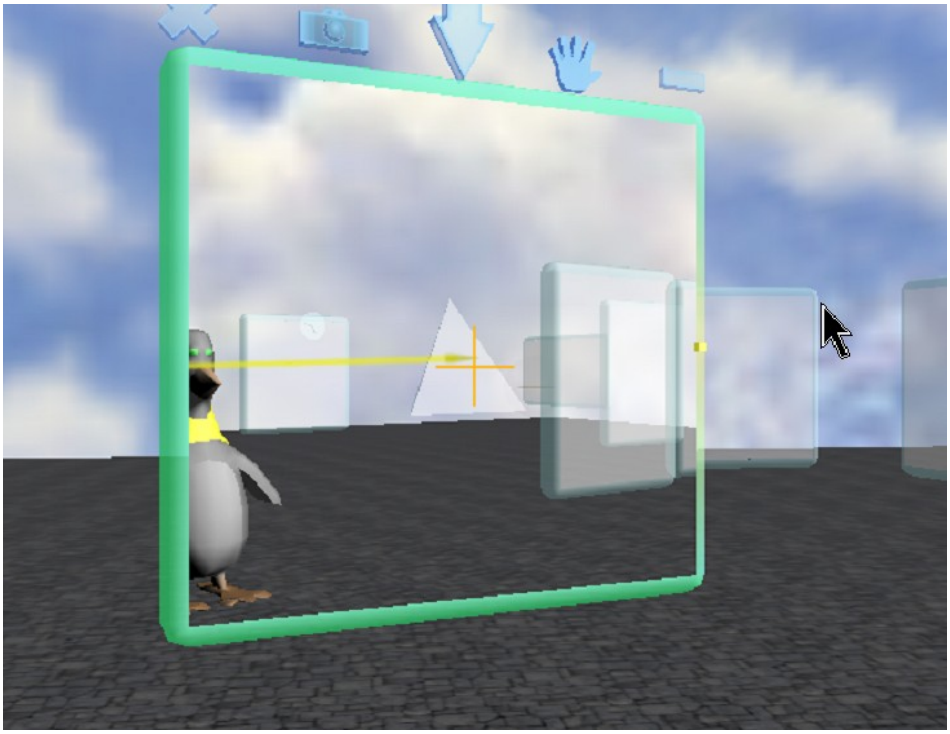


Click on the top window frame and drag it in order to move the window in a plane perpendicular to line of sight of the user.

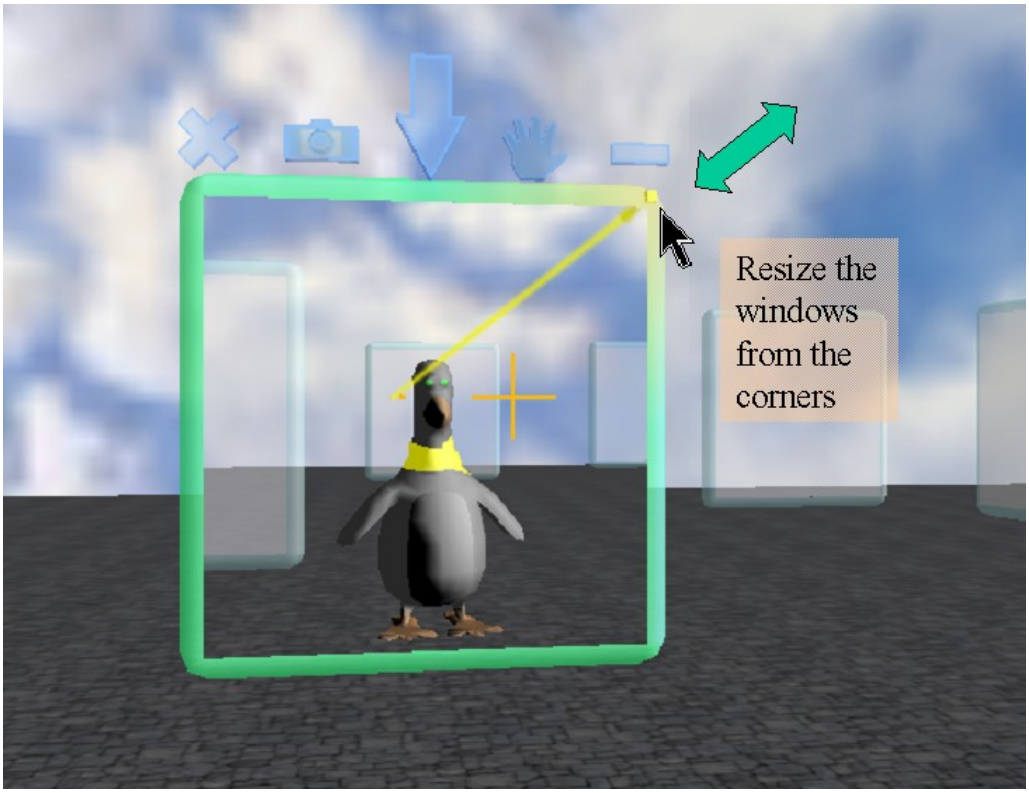
Click on the bottom window frame to drag it in the plane of the ground.



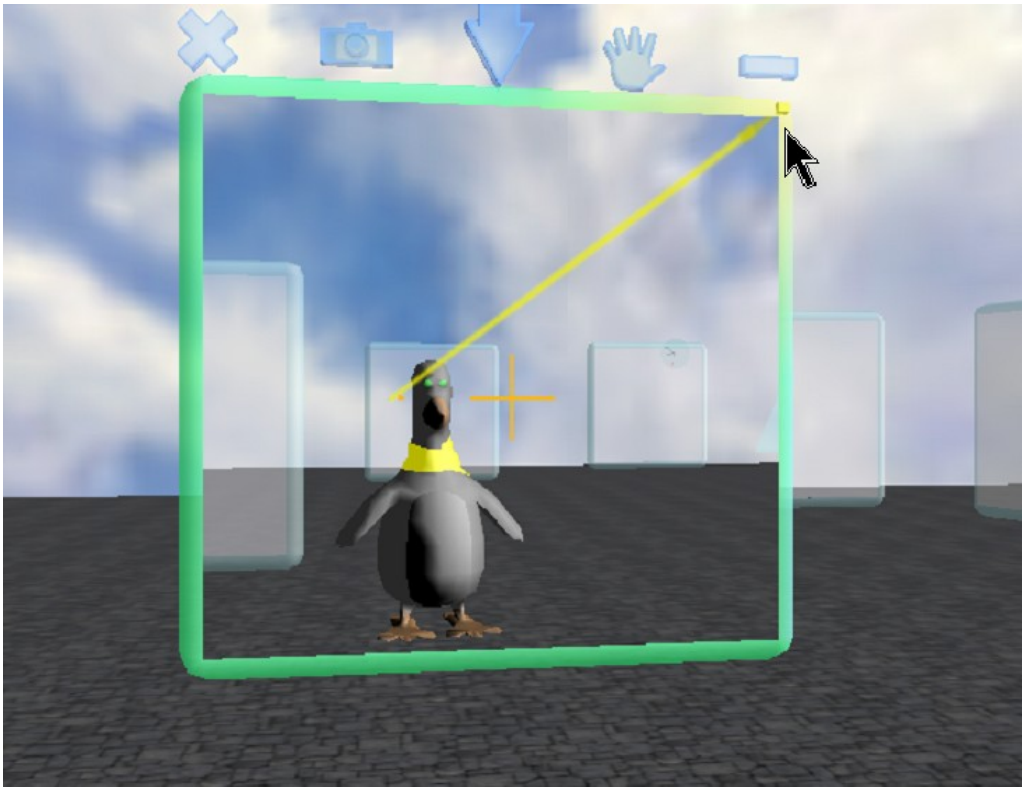
Click on either side frame of the window and drag horizontally in order to rotate the window about the vertical.



The result of rotating the window.

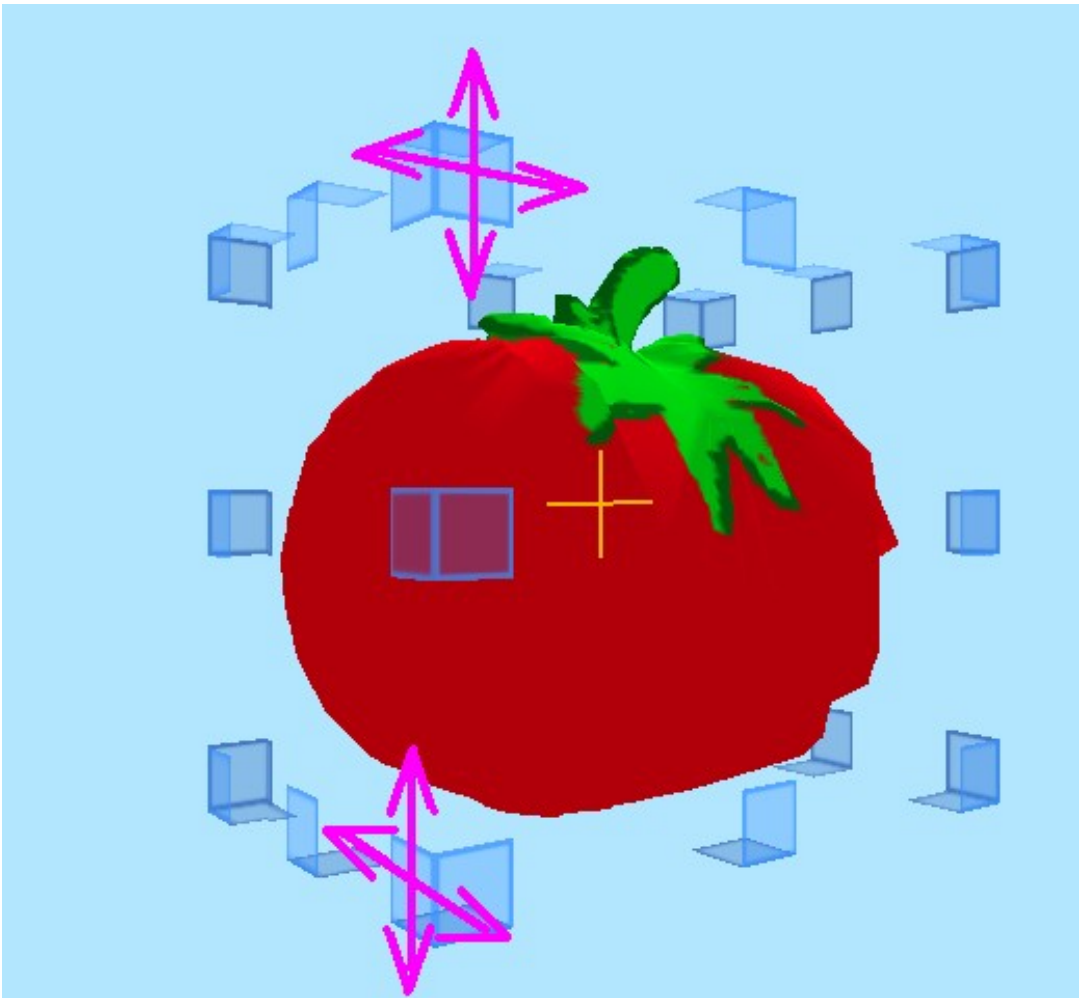


Click on any corner of the window and drag to resize it.



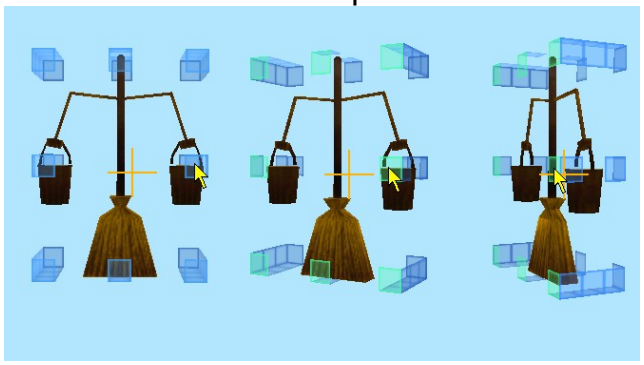
3D Edit Box

The 3D edit box is used to manipulate and position objects in the 3D environment. It is used to translate the object parallel to any of the faces of the box, rotate around any of the three axes, and scale it in and out from the center of the box. The side of the box is made up of eight squares, four in the corners, and four on the edges. Three sides meet at each corner of a box, so in each corner we see three rectangles. Two sides meet at each edge of a box, so we see two rectangles meeting at each side of the box.

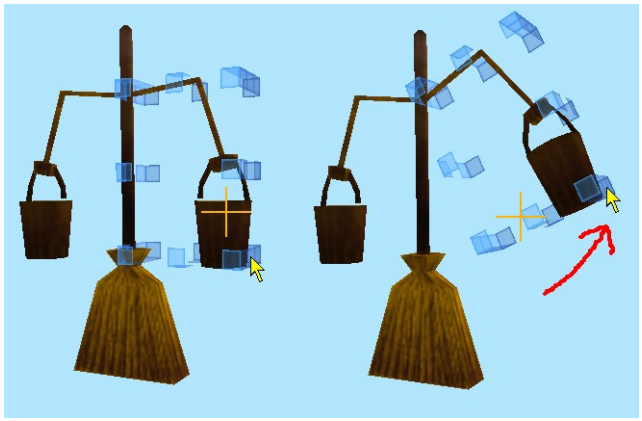


In this image, we see the edit cube. If the user selects one of the corner rectangles in the edit cube, the object will be translated in a direction parallel to the surface of that corner rectangle. The arrows above indicate the directions that the object can be dragged when the underlying rectangle is selected. This allows the object to easily be positioned virtually anywhere in the 3D space. This utilizes the planar translation primitive interface control.

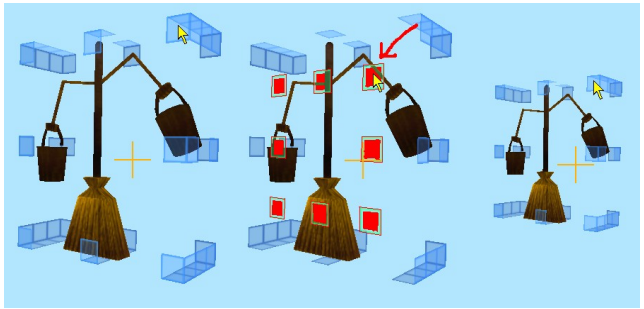
When the user selects one of the edge rectangles, as shown, the object is rotated around the axis parallel to the selected edge.



This rotation is performed in the frame of reference of the object itself. This makes it trivial to use. This rotation uses the cylindrical spinner primitive interface control.



Complex, hierarchical objects can be easily edited as well. Once the object is selected, the children of the object can be selected by pressing a down arrow. Sibling objects can be selected by pressing the left or right arrows, and the parent can be selected by pressing an up arrow. Once selected, each of the individual children can be edited in exactly the same way as the above picture illustrates.



The corner rectangles can also be used to scale the object. If the corners are selected while the shift key is down, then the object is scaled proportionately by the distance of the cursor from the center of the edit cube.
One of the key problems with the earlier effort was that if the enclosed object had any functional elements, they were unavailable while editing. The Croquet edit box has all of the editing capabilities of the earlier effort, but allows complete access to the enclosed object's function.

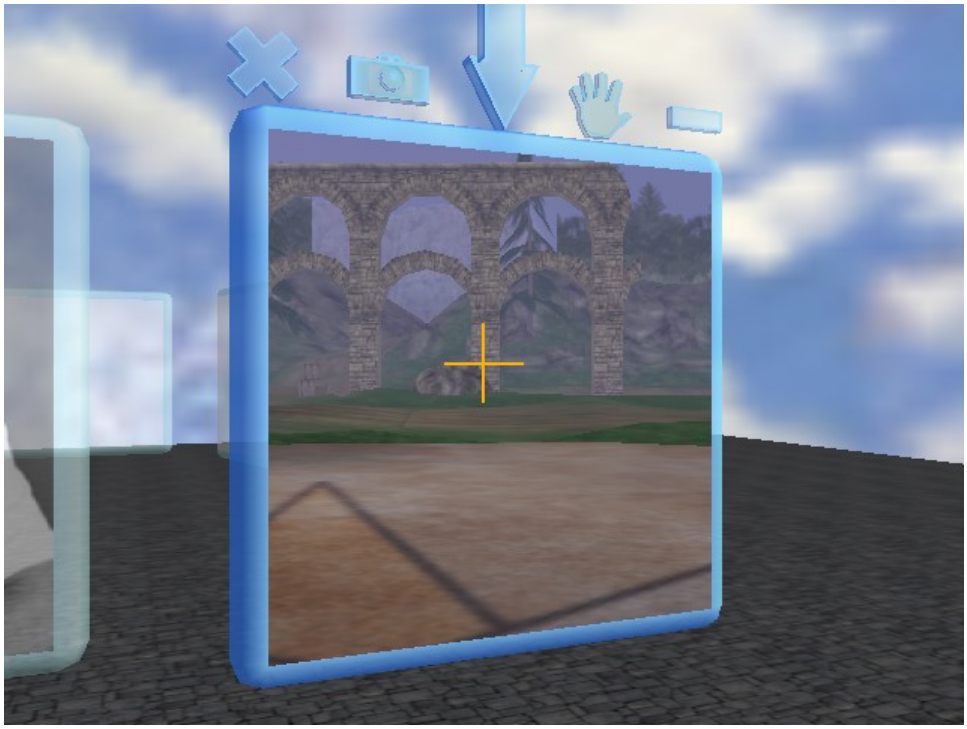
Portals, Postcards, and Projections

Simply put, a Space is a place. In Croquet, a space is a container of objects, including often, the user's avatar. A good example of a space might be a child's play room. All of his toys are objects that happen to be lying on the floor, or perhaps put away. A child can always come into the room to play, or even to pick up a toy and carry it outside. In Croquet, Spaces can act like rooms, but they can also act as landscapes, or virtual conference rooms, or any kind of 3D container of any size.

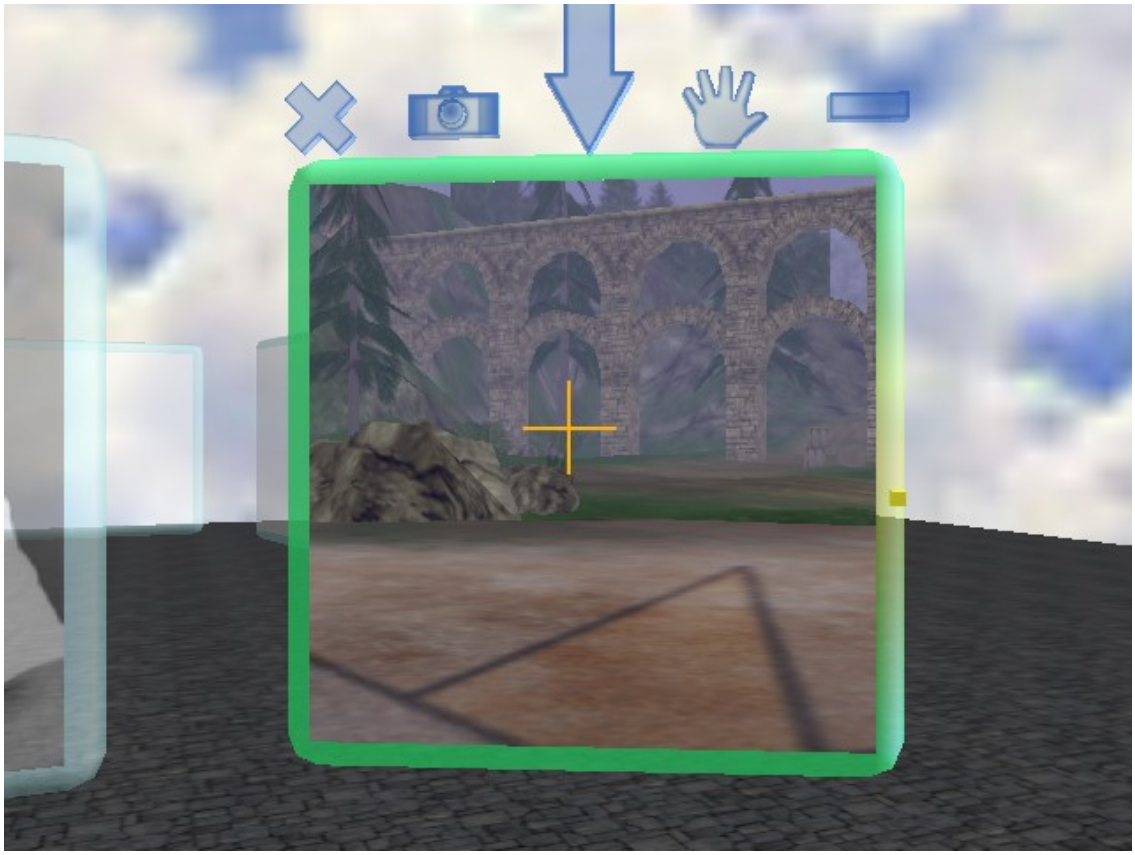
2D Portals

Portals are simply a 3D spatial connection between spaces. If you place one portal in one space, and a second portal in a second space and link them using the other's TPostcards, then you can view from one space into the other. In the example of the child's room, a portal is simply the door to the hallway. The hallway is just another space. One key difference between Croquet portals and spaces and the real world is the concept of actual versus virtual location. In the real world, the hallway must be physically next to the child's play room, or the door simply won't go anywhere – at least it won't lead into the hall. In the virtual world, a portal can connect ANY two spaces, even if one is located on a computer half a world away. Physical location doesn't mean anything. Connections are all virtual. Consider as an example, the mirror. In Croquet, a mirror is actually a portal that happens to be linked back to itself. In other words, it is actually a door that happens to open into the room from which it is leaving.

A closed Portal into another space – ready to be opened…



The Portal is now open and we can see into the linked Space, in this case the entrance to a multiplayer game.



Here the portal has been rotated toward the user. Just like the mirror, we have a slightly different view into the game world.

One of the key aspects for Croquet is the ability to have a portal dynamically move around in a space, while allowing the proper view through the portal. This is a bit strange, but it works like this: when you look through a window, what you see is determined partially by your position relative to the window. If you move to your left, you can see more of the space to your right (and vice versa). But, if you could pick up the window and move it relative to your position - instead of you moving relative to its position, the exact same thing should happen. It should be much like picking up a box and looking through a hole in it. You turn the box around to see different areas.

The big win for portals is that they allow the user to jump from one virtual space to another by simply walking through the portal, just as the child walked through the door from his play room. What is different in Croquet is that the portal can lead to anywhere in the virtual world. In turn, portals that are contained within these spaces can themselves lead to other worlds.

This sequence shows a remote avatar jumping through a portal into a virtual game world.

Portals have a number of important uses.

Portals allow a world designer to partition a space into connected areas; this is now a standard part of any reasonable game engine. If a portal is not visible from a particular position and orientation, none of the objects inside the portal are rendered. This is essential for complex spaces.

We can use portals for project management. They allow a hierarchical structure (or any other kind of graph) for managing worlds. They act the same way that a project window in Squeak or live folders on a desktop might work.

Portals act as virtual links between users spaces. The user will be able to use it as a bookmark into another user's workspace and access it at any later time.

Islands and their contents can only be accessed using a view portal and a TPostcard.

TPostcard

The idea of a TPostcard is very similar to that of a URL link in a web browser. It specifies a location somewhere on the net where the linked content may be found. In our case, a TPostcard links to a Croquet router, which in turn references a replicated Island. It goes further than that, we also need to know which particular object inside of the Island we are referencing, and sometimes may even need to know if there is a transformation associated with this link.

The TPostcard is used to find the Router in the following way:

Look up the routerAddress. If this is a valid address, it will initiate a connection. If it is nil, it will look for the TContact matching the routerName/ routerID, and then connect to the router specified by that TContact. The set of known TContacts is automatically propogated over the local area network, can also be populated by the programmer.

Once the router has been discovered, the Island is synced.

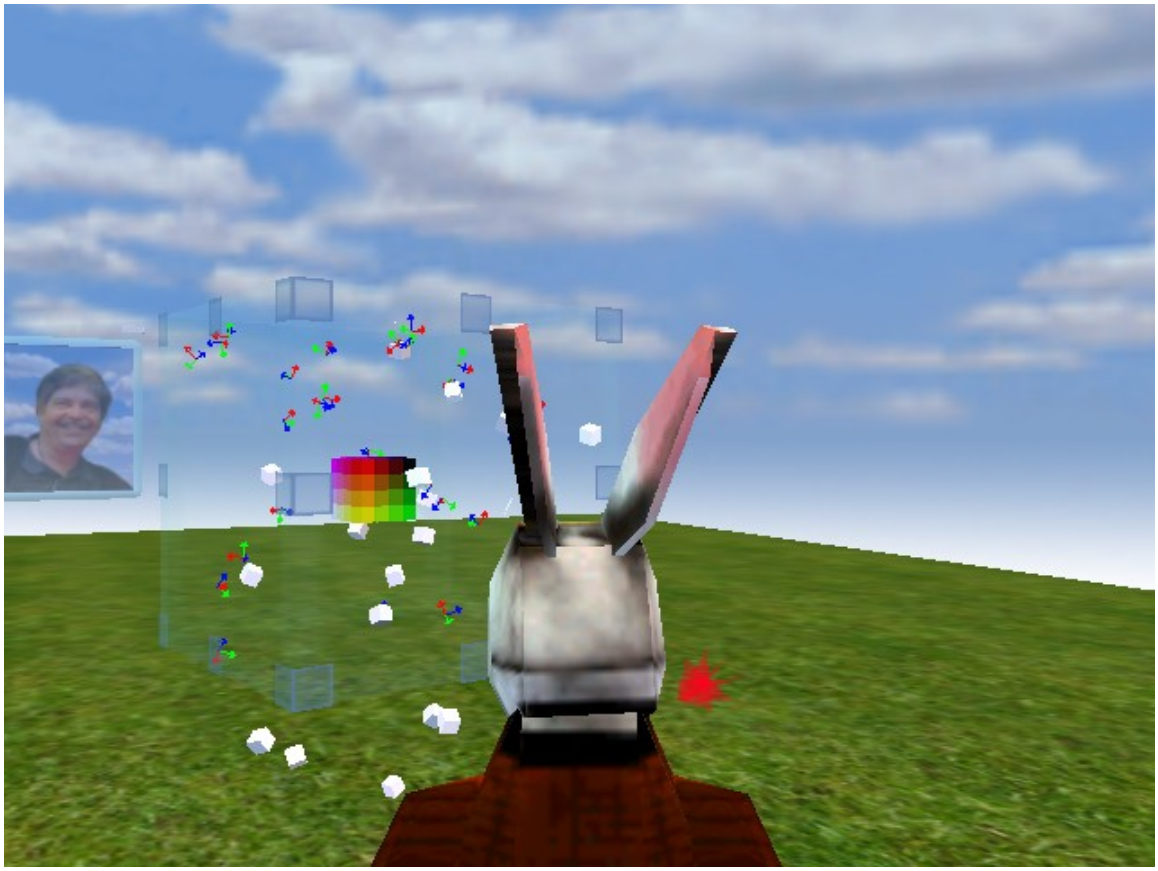
These instance variables MUST be left as references. Every element in them must exist inside of an Island that may be replicated. There can be no controllers or FarRefs placed into this object. The standin must be constructed by the containing Island.

routerAddress – the address of the router on the net.
routerID - the routerID is a unique ID associated with a single Island with which it should share the same ID and name.
routerName - the routerName is a user created name that can describe the Island/Router group and help the user find the router again.

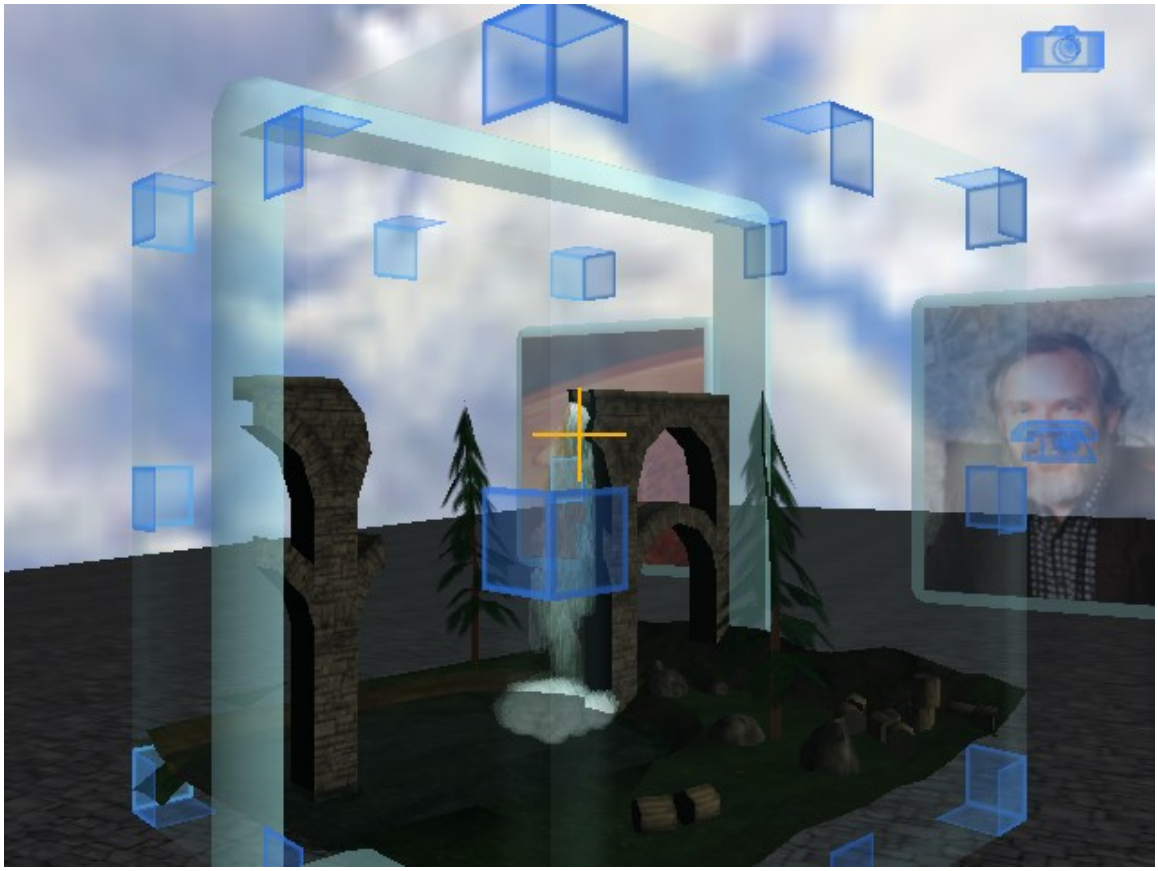
viewpointName - the name of an entry point into the island. This will typically be a TFrame of some sort.
viewpointID - the ObjectID of the entry point.

3D Portals

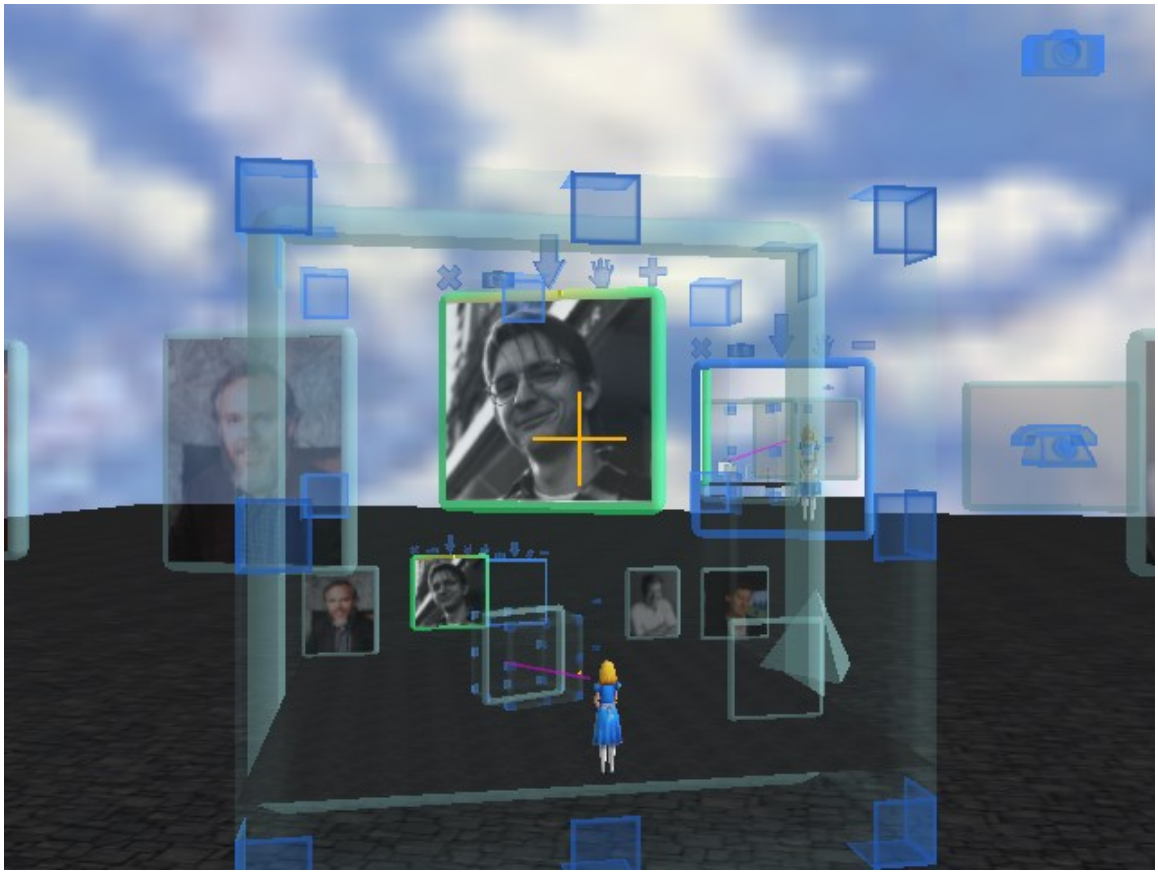
3D Portals (TPortal3D) are just a different kind of portal – one where we can view and manipulate the contents of another (or even the same) space. In the example below, there is a TScrollBox3D around the 3D Portal. The handles of this box allow the user to scroll around inside of the micro space, rotate the portal around any axis, and drag the scale the contents of the TPortal3D.



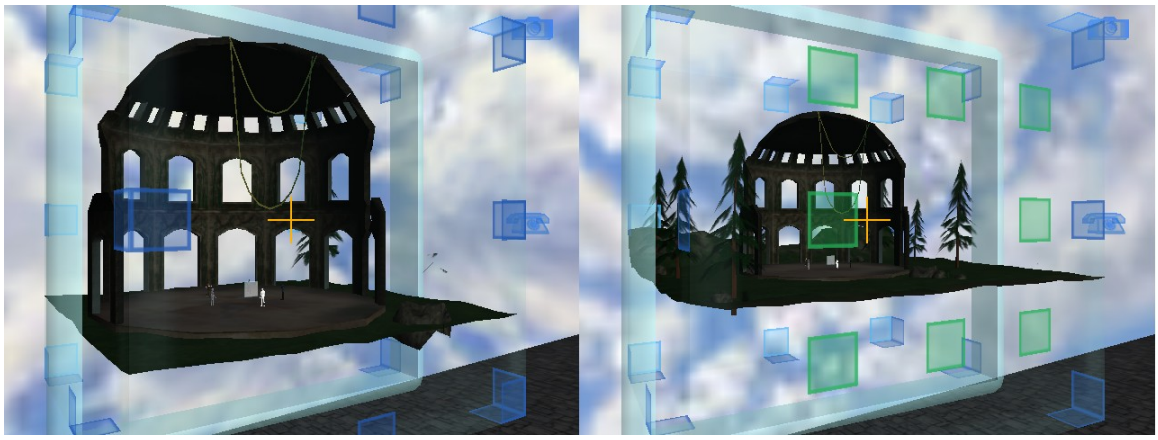
3D portals are complete miniature copies of existing spaces that are completely accessible and editable by the user. They are inside of a clipping box, which can in turn be inside of another 3D edit box. 3D portals are extremely useful for getting a complete overview of an environment, even when the user is inside that same environment.



This is an example of a 3D portal inside of an edit box. The edit box has the same properties as described above, except that instead of translating the entire contents of the box, the corner rectangles are used to scroll through the 3D space.



Here we see a miniature version of the same space that the user is in. Notice that the windows inside the smaller 3D portal are identical to the outside environment. The girl in the blue dress is actually a miniature version of the user's avatar. The user is moving the window in the center of screen by manipulating it in the 3D portal, but the full-size version is moving in exactly the same way. This is simply because it is really the same object.



The above image demonstrates how the 3D portal allows the world can be scaled using the 3D edit box scaling rectangles.

A key advantage of 3D portals is that because they are always live working environments, they allow for a god's eye view editing capability that is simply unavailable in traditional approaches to design. The user can edit a miniature version of an object in the space while experiencing the results of the edit in the full-size space.

The View Portal

Even the Croquet view screen is actually a TPortal. This is basically a TPortal that is used by the camera to render a scene, only instead of having it appear as a link between spaces, it is a link between the user and the space.

Overlay Portals

Any TPortal can also have other TPortals as "overlays". These are TPortals that are linked to spaces that may contain user interface objects, or may be used to extend the functionality of an Island with a "ghost" overlay. A "ghost" overlay portal makes the content in that overlay look as if it is just another part of the world inside the main space, including giving the user access to modify and control these ghost objects.

System Overlay

The System Overlay is just a standard Croquet overlay that is designed to work with the View Portal. It contains standard Croquet user interface objects, and any additional objects the user may wish to add. The System Overlay is not replicated.

Projections

A Projection is something like a TPortal, but instead of rendering an entire space from a particular point of view, a Projection will render a remote Island TFrame and its children as if it is inside of the local Island being rendered. This allows objects to be moved between Islands extremely efficiently, and allows a single TFrame object in its own Island be useful to a number of users in a number of different contexts at a time.

Typing and Key Entry

Some objects react to the keyboard when they are highlighted by having the mouse pointer over them.

Connecting via LAN and WAN

Croquet is a toolkit for building real-time collaborative systems. To really experience Croquet you should create a shared, collaborative space inhabited by more than one user. You will probably want two computers on a local area network to try this. Here is how:

First, launch the Croquet SDK, then click on the "First Steps" button on the splash screen. You can then drag a Croquet world object out of the green objects bin to create a live instance of Croquet. We describe two different ways of establishing a shared space below.

Master/Participant Connections

Join a common space using Demo (Master) and Demo (Participant).

1. On one computer, drag the Demo (Master) object out of the objects bin. Croquet will build an instance of the demo world, and make it available for participants to join. The master is ready to be joined once the clouds in the sky start moving.

2. On another computer, drag the Demo (Participant) object out of the objects bin. The participant object will look for a master on the LAN, and after it finds one, it will synchronize itself by copying the state of the shared world, and then begin running it. You will be able to tell you have synchronized with the other user(s) when you see the world and the clouds begin moving.

3. Now that you and the other user(s) have joined the common world, you will be able to see each other's avatars, and changes made in the world such as moving or creating objects, are reflected in everyone's copies of the world. If one of the participants leaves the world, their avatar will disappear, but they or other users can rejoin the world.

Master/Master Connections

Establish a connection between two different master spaces using Sailing (Master) and SimpleDemo (Master).

1. On one computer, drag the Sailing (Master) object out of the objects bin. Croquet will build an instance of the boating world, which will then be available for participants to join.

2. On another computer, drag the SimpleDemo (Master) object out of the objects bin. Again Croquet will build an instance of this world and make it available for others to join.

3. From either of the worlds, go to the Talk>Chat menu, and select the "Connect to Another World" option. In the Postcard settings window, select from the popup list of nearby worlds, then click the OK button. This will create a one-way portal to the other world. If you then click in the content area of the portal window to open it, you will synchronize with the other world, and will be able to drive your avatar through the portal to enter your friend's world.

4. Once you are in the shared world, you and the other avatar can collaboratively update the world. To coordinate your actions, you can use the text or voice chat tools found in the talk menu.

Currently, the connection mechanism is somewhat hard-coded. The system works extremely well once the connection is made.

We create a Master Island, there should be only one on a network, and any number of participants. There is a simple broadcast discovery mechanism for local area networks, where all of the routers associated with a master broadcast their existence and location to the other participants. If the participant's setup has specified an interest in that particular Island, then a join is automatically requested and the Islands are synchronized. Typically, we have labeled the morphs that generate a new master Island/Router pair as Master, and the participant as Participant.

For wide area network connections we currently need to set this up by hand. Code illustrating how this is done is in the section below called Connecting. We are developing discovery services that will dramatically simplify this process. Ultimately, this will be as easy as specifying a page location in a browser.

Croquet System Overview

Coquet is a new approach to developing and delivering collaborative interactive media applications. Every part of the system is designed around enabling real-time, identical interactions between groups of users. The architecture of Croquet actually makes it quite easy to develop collaborative applications without having to spend a lot of effort and expertise in understanding how replicated applications work. There are a number of simple patterns and rules to remember, but otherwise, it is quite simple to quickly develop very powerful systems. This section is a brief overview of how the Croquet system works, but if you want to simply dive into creating your first applications, then skip over this to the chapters on “Creating New Croquet Islands” and “Creating New Croquet Objects”.

TeaTime and Islands are the basis for Croquet's replicated computation and synchronization. They are designed to support multi-user applications that can be scaled to massive numbers of concurrently interacting users in a shared virtual space. Croquet’s treatment of distributed computation assumes a truly large scale distributed computing platform, consisting of heterogeneous computing devices distributed throughout a planet-scale communications network. Applications are expected to span machines and involve many users. In contrast with the more traditional architectures we grew up with, Croquet incorporates replication of computation (both objects and activity), and the idea of active shared subspaces in its basic interpreter model. More traditional distributed systems replicate data, but try very hard not to replicate computation. It is often easier and more efficient to send the request for the computation to the data, rather than the other way around, however. Consequently, Croquet is defined so that replication of computations is just as easy as replication of data alone.

Islands and Replicated Computation

The basic unit of replication and collaboration in Croquet is called an Island. A Croquet Island is first a secure container of arbitrary objects. Access to the contents of an Island is governed by strict rules that ensure proper bottlenecking. Internally, objects see no difference between the rights they have in accessing other objects than in any other traditional system – except for a significant restriction that explicit infinite loops are simply not allowed. This is replaced instead by a mechanism we refer to as “temporal tail recursion”, discussed later.

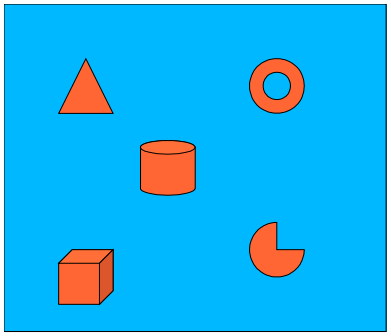
Croquet is based upon the concept of replicated computation – rather than replicated data. Croquet is based upon a synchronized message passing model, where the messages themselves ensure that the replicated systems remain consistent between machines. Though it is necessary to synchronize the world

state of a new user by transferring the current contents of the world, after that, the worlds stay consistent only through the creation and processing of time based messages.

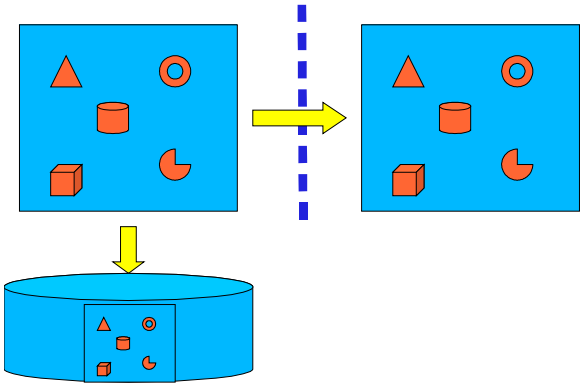
Islands are the units of replication. A single Island is actually made up of a collection of containers all in identical states on different machines connected by a network. One of these containers on a particular machine could be thought of as a replica or projection of the Island - albeit a complete projection. Croquet guarantees that the evolution of the state of a particular Island replica is identical to any other replica of the same Island. This is the basis of the Croquet collaboration architecture.

The term “Island” is used in several ways in this document. As just described, an Island consists of the set of all of its replicas. For clarity, we may refer to a copy as an “Island replica” (or simply “replica”), while the whole set is called the “Island”. Furthermore, Island is a Squeak class that implements the base “Island” model; a replica is an instance of the Island class.

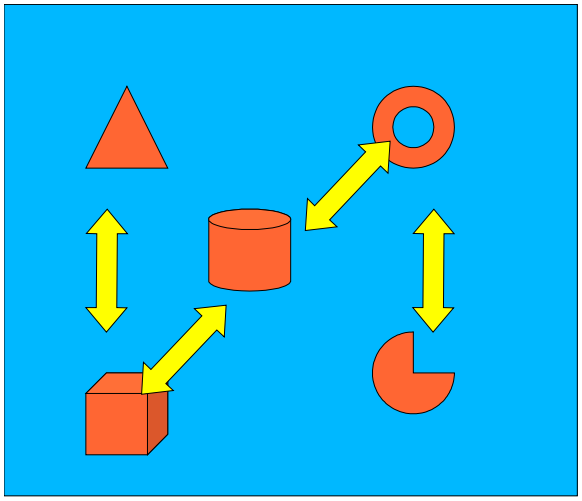
Croquet Islands



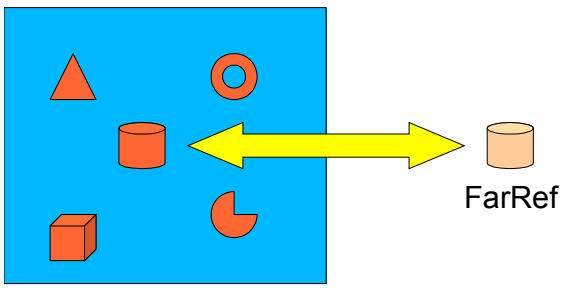
Croquet Islands are secure containers of other objects. They act as a kind of meta-object in that they have perhaps an even better model of encapsulation - certainly more secure, than traditional object models and they enforce a rigorous content-hiding and message passing model. This is a necessary precursor to guarantee identical behavior and identical response to external events. [A similar concept to Islands is the Vats in the E programming language.]



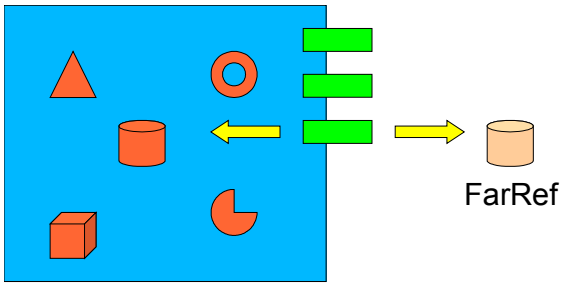
Croquet Islands are the basic unit of replication. They are generic object containers that are simple to checkpoint and exchange. They can easily be saved to disk for use later or archiving, or they can be transported between users to initiate collaborative interaction with the contents of the Islands.



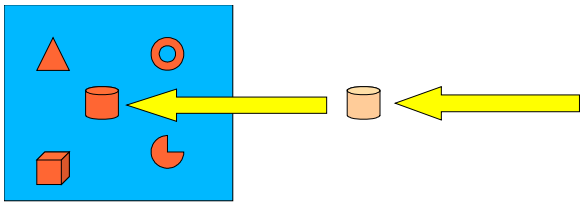
Objects internal to an Island have the same access privileges to each other that ordinary objects have. They can send messages directly to each other or themselves, can maintain direct access links to each other, and in general exhibit the same kinds of relationships that ordinary objects enjoy. They cannot, however, send messages outside the scope of the Island – nor can objects outside the Island send messages directly to the objects inside.



That is not to say that there is no way for external messages to be sent to an object inside of an Island. A TFarRef is an object that exists outside of the Island, but can act as a proxy for an object that is actually inside the Island. A message sent to the TFarRef is forwarded on to the actual object that is inside. This is actually a somewhat simplistic view of what actually happens – the actual process is a bit more interesting.



The TFarRefs are actually generated by having the Island register a particular object as being externally accessible. An external name is generated, and a TFarRef is made available by the Island. The Island maintains a Dictionary that maps the TFarRef back to the original object.



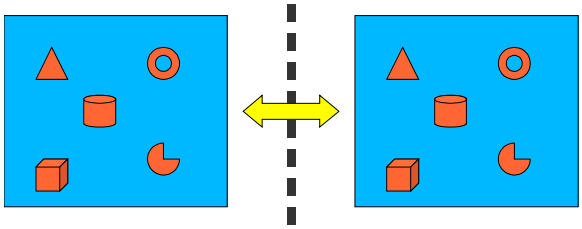
In Croquet, messages are sent to the original object inside of the Island indirectly via the TFarRef. This ensures that we have a nice way of bottlenecking the message, as we will usually have to redirect it in such a way that ensures it is properly replicated.

Though there are ways to bypass this bottlenecking, it is extremely dangerous do so, as it can easily lead to a violation of the replicated state of the system. This invalidates the guarantee the Croquet has of ensuring perfectly replicated state inside of an Island. However, certain actions simply cannot be performed in a replicated way. An example of this is rendering of the content of an Island. Rendering only makes sense to a local observer, and is a relatively expensive action. Replicating the action of rendering a scene in Croquet is not only inefficient; it also does not make much sense.

Replicated Islands

Islands in Croquet are the units of replication in the system. For Islands to work properly, they must be deterministically equivalent. This means that given an identical initial state between two Island replicas, and given exactly the same

inputs at the same time – the end states of these Island replicas must be identical. If for some reason there is even a slight divergence in state, this can easily be multiplied such that the end results are completely out of sync. Since the entire point of Croquet is to provide the users a perfectly replicated simulation environment that can be used as the basis of communication of information and ideas, this kind of breakdown renders the system totally useless.

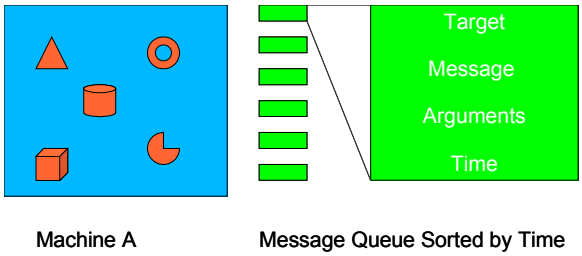


Of course, the entire point of the Island architecture is to have any number of Island replicas exhibiting identical state anywhere on the network – hence, anywhere in the world or even beyond. A number of new concepts and objects need to be introduced to describe how the replicated Island architecture works.

The first is the Croquet Message, which includes not just the token message name, but the target of the message, the message arguments and when the message will be executed. The second is the Croquet Router, which is the object that manages messages that are generated externally to an Island but are sent to it. It both determines when this external message will be executed and replicates it to all of the Island replicas. The third is the Croquet Controller, which is the interface between the Island and the Router and manages external events by redirecting them to the Router. Together with Islands, these are the main elements of a robust time-based replicated architecture.

Croquet Messages

A Croquet message is made up of four components, the target – which is the object that will actually execute the message, the actual message, the arguments to the message, if any, and the time at which the message will be executed. The time value is also used to sort the unexecuted message in the Island's message queue. Croquet messages can be generated either internally, as the result of the execution of a previous message inside of an Island, or externally, as the result of an external event usually generated by one of the users of the system.



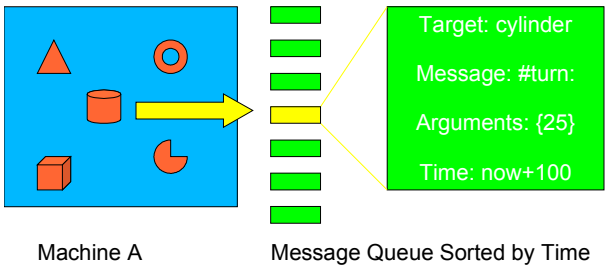
An Islands message queue and a message.

There is virtually no difference between internally and externally generated messages as far as the internal execution of the Island is concerned. A major difference between the two is that the timestamps on externally generated messages are used by an Island to indicate an upper bound to which the Island can compute its current message queue to without danger of computing beyond any possible pending messages.

Timing is everything

The definition and manipulation of time plays the central role in how we are able to create and maintain a replicated Island state. We must be able to guarantee that every internally generated message will be executed in exactly the proper order at exactly the proper time. Externally generated messages must be properly interleaved with the internally generated messages at exactly the right time and order.

When a new message is generated, it is inserted in the sorted queue based upon its execution time.



A new message inserted into the message queue.

Island Time

An Island's view of time is defined only by the order of the messages it has in an internal queue. Islands can only respond to external, atomic, time-stamped messages. These messages are literally the Island clock. Though Islands have internal time based messages that can be queued up, these cannot be released for computation until an external time based message has been received which indicates the outer temporal bound to which the Island can compute to. This is a key point of the architecture. Though we may have a huge number of internal

messages ready to be executed, they remain pending until an external time stamped message is received indicating that these internal messages are free to be computed up to and including the newly received message. Each Island's message queue is processed by a single thread, so issues with improperly interleaved messages do not arise.

When a message is executed, the time remains atomic in that it does not advance during the execution of this message. The "now" of the message stays the same. When we generate a future message during the current message, we always define its execution time in terms of the current "now" plus an offset value. This offset must always be greater than zero (though in fact zero is an acceptable value in certain circumstances, it should almost always be avoided because if it is infinitely iterated, Croquet can't advance and the system will appear to freeze.) If we generate multiple future messages, they will have an identical "now", though they may have different offsets. If we generate two messages at the same "now" and with an identical temporal offset value, an additional message number is used to ensure deterministic ordering of the messages.

All of the messages in the Island queue are "future" messages. That is, they are messages generated as the result of the execution of a previous internal message with a side effect of sending messages to another object at some predefined time in the future, or they are messages that are generated as the result of an external event – usually from a user, that is posted to the Island to execute at some point in the future, usually as soon as possible. All of these messages have time stamps associated with them. The internal messages have time stamps that are determined by the original time of the execution of the message that initially posted the message plus the programmer defined offset. The external messages have a time that is determined by an external object called a router and is set to a value that is usually closely aligned with an actual time, though it doesn't need to be.

Internal future messages are implicitly replicated; they involve messages generated and processed within each Island replica, so they involve no network traffic. This means that an Island's computations are, and must be, deterministically equivalent on all replicas. As an example, any given external message received and executed inside of a group of replicated Islands must in turn generate exactly the same internal future messages that are in turn placed into the Islands' message queues. The resulting states of the replicated Islands after receipt of the external message must be identical, including the contents of the message queues.

External future messages are explicitly replicated. Of course external messages are generated outside of the scope of an Island, typically by one of the users of the system. The replication of external messages is handled by an object called

a Router, which in addition specifies when the message will be executed. The Router is more fully described below.

External non-replicated messages are extremely dangerous and must be avoided. They do play a role, but it is extremely rare that anyone will ever have a need to make use of this mechanism. The problem is obviously that if a non-replicated message is executed and happens to modify the state of an Island it breaks the determinism the Island shares with the other replicated copies. We do use such non-replicated message when rendering the contents of an Island, but this is extremely well controlled.

Each Island has an independent view of time that has no relationship to any other Island (Island used here as the complete collection of Island replicas). This includes (for example) that a given Island could have a speed of time (relative to real time) that was a fraction of another. This is useful for collaborative debugging, where an Island can actually have a replicated single step followed by observation by the peers.

Since time is atomic and the external messages act as the actual clock, latency has no impact on ensuring that messages are properly replicated and global Island state is maintained. It does mean that higher latency users have a degraded feedback experience.

Islands enforce an internal "temporal tail-recursion" with the use of the #future: message. Basically, a message is arranged to execute some unit of time from the atomic "now" in the future. Hence, a message like:

```
#turn: angle
        cube rotateAroundY:angle.
        (self future:100)turn: angle+1.
```

causes the angle to be increased by one degree every 100 milliseconds.

The Croquet Router/Sequencer

The Croquet Router plays two major roles. First, it acts as the clock for the replicated Islands in that it determines when an external event will be executed. These external events are the only information an Island has about the actual passage of time, so the Island simply cannot execute any pending messages in its message queue until it receives one of these time-stamped external messages. The second critical role played by the Router is to forward any messages it receives from a particular Croquet Controller to all of the currently registered Islands.

Given that Islands cannot execute beyond these external messages, it is usually necessary to manufacture new messages simply for the sake of moving time forward. These messages are created by the Router and are called heartbeat messages. They are basically message-free and contain only a time-stamp that allows the island to execute to.

Routers can be located almost anywhere on the network and need not be collocated with a particular Island. Typically, the creator of the Island will own the Router by default.

The Croquet Controller

The Croquet Controller is the non-replicated part of the Island/Controller pair. The role of the Croquet Controller is to act as the interface between the Island and the Router and between the user and the Island. Its main job is to ship messages around between the other parts of the system.

The Controller also manages the Islands message queue, by determining when messages will get executed.

Interestingly, a Croquet Controller can exist without an Island, acting as a proto-Island until the real island is either created or duplicated. In this case it is used to maintain the message queue until either a new Island is created or until an existing Island is replicated.

Croquet Message Execution

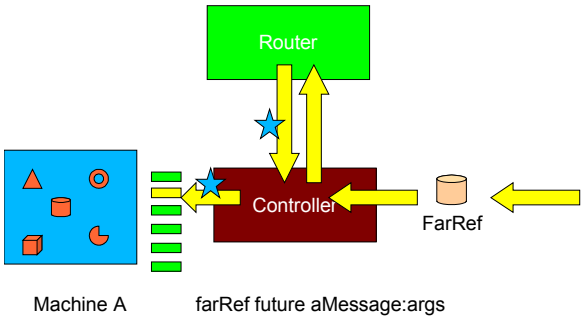
The basic idea behind Croquet's replicated message model is that the Croquet Router acts as the clock for all of the Island replicas. This is a guarantee that they all share exactly the same model of time. The Croquet Controller acts as the interface between the Router and the user and the Router and the Island. Every replica of an Island has its own Controller, but there is only one Router for the set of replicas of an Islands.

To track a message from an initial event to execution inside of an Island, we first consider a user interacting with a specific object. The user never has direct access to the objects inside of an Island, so he can only interact with a far reference to that object, a TFarRef. A message is constructed using the following line of code:

```
farRef future aMessage:arguments
```

What we are doing here is sending aMessage: to the farRef to be executed as soon as possible in the future. In fact, another way to read #future is "ASAP". The farRef forwards this message to the Croquet Controller of the Island that

contains the actual object that the farRef refers to. The Controller simply forwards the message again to the Croquet Router associated with the Island. The Router immediately places the current time stamp on the message – note that no two time stamps are equivalent – and forwards the message, now containing the execution time, back to the original controller. The controller then inserts the message into the Islands message queue and begins to execute all of the messages that are already in the queue that have a time stamp less than the new message.

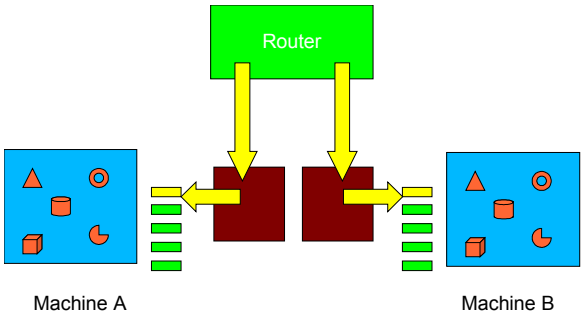


This may seem a bit round about just to get a message to the local Island, but this process makes much more sense when seen in the context of replicated messages as described in the next section.

An interesting thing to point out is that a given internal message can generate a new message at a delta from the time of the original internal message that is actually less than the time of the new external message that is driving the clock. This newly minted message is simply added to the queue and executed in its own proper order before we actually execute the external message. If in turn its delta is small enough, it may even generate a number of additional messages that get executed before the external time stamped message is.

Replicated Message Execution

The main reason for the existence of the Islands, Routers, and Controllers is to enable the perfect replication of even complex interactions and simulations. The model for this is basically identical to the description in the previous section up to the point where the no time-stamped messages are sent out of the Croquet Router. A replicated Island has multiple identical copies of itself in various locations around the world (or the office or school). There is still only one Router, but now, for every Island replica there is a Croquet Controller.



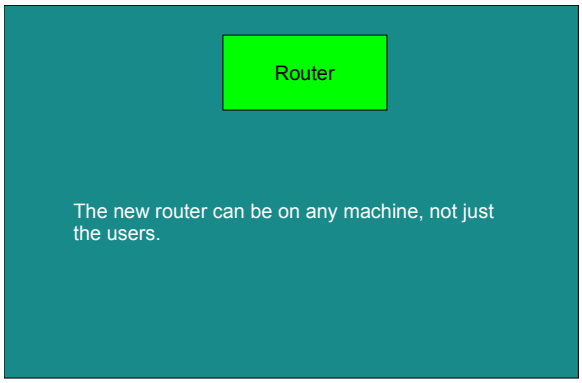
After the Router receives a message from one of the Controllers, it now forwards the time stamped message to all of the Controllers connected to it – including the original one. The Controllers all insert the new message into the sorted message queues and executes the messages in each queue up to and including the new message. This means that every Island is now completely up to date with every other one.

Starting, Joining, and Participating

The process of creating a new Croquet session from scratch and then having new users join into the process is relatively simple. There are three parts to it – creating the Router, Controller, and Island – Joining the Router – and Participating in the Croquet session.

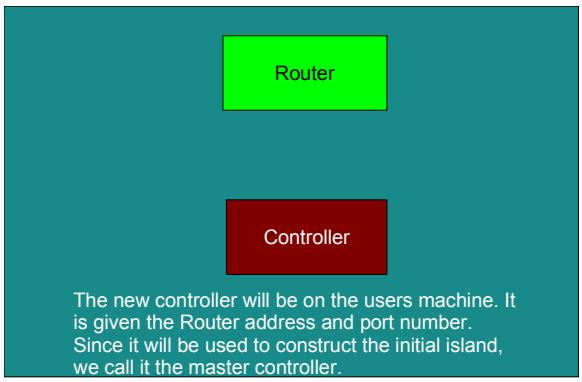
Starting Up

The first action required in creating a new Croquet world is for us to create a new Croquet Router. This Router can be on any accessible machine on the network – either remotely on a WAN, locally on the LAN, or on the same machine that will act as host to the original Island. Routers are extremely lightweight objects, so they really don't take up many resources, either in space or computation. The Router has a network address and port number that is how we will find it later.



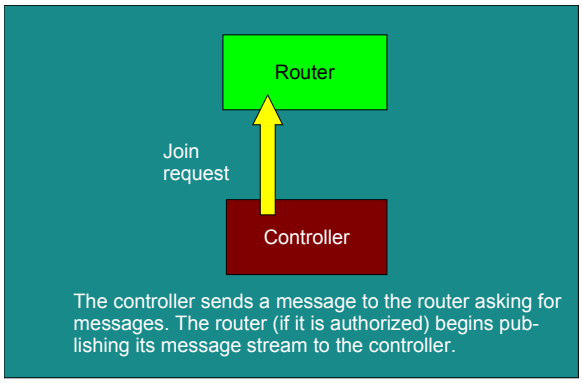
Once the Router exists, we need to create a new Croquet Controller. This needs to be located on the same machine that the new Island will be located

on, and is usually the original users own computer. Again, this is not essential. It is quite easy to create a Croquet Controller/Island pair on a remote server. Of course, this may take a few more resources than a simple Router requires. We give the Controller the address and port number for the original Router and it begins to connect.

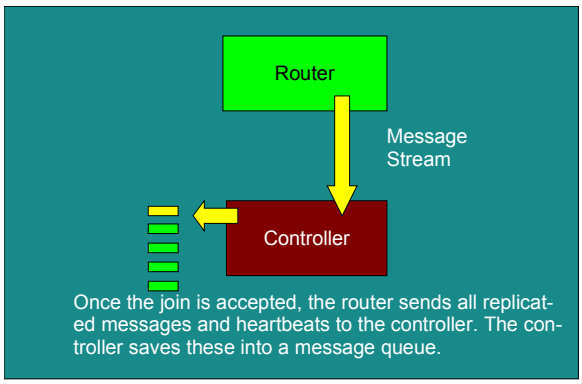


Joining

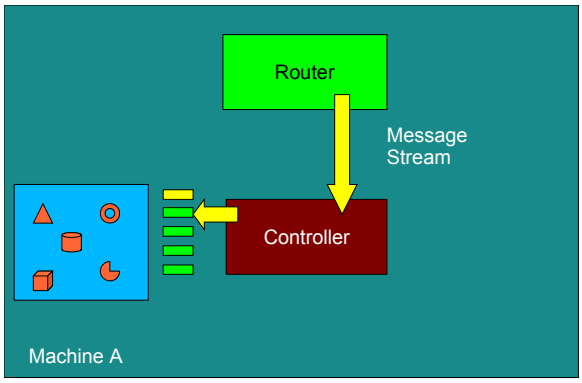
The first thing the Controller does is send a message to the Router asking to subscribe to its message stream. Given that we made both the Router and the Controller, we are guaranteed of getting access – but it is important to note that this may not be true for other users as they attempt to join. You will have to grant them explicit permission – or leave the Router open to anyone, if they are to join the session. Once the Router authorizes the Controller it will begin publishing its message stream to it.



The only messages coming from the Router at this point are the heart beat messages – assuming we set the Router to generate these. In any case, the Controller is designed to simply begin adding these messages to its message queue. This is actually important when we are joining an already existent replicated Island, because in that case many of the messages that get sent and stored on the queue will be necessary to bring the Island replica up to date after it is replicated locally. Joining is view only access. At this point, even if there were an Island, the user is not allowed to send messages that might modify it in any way.



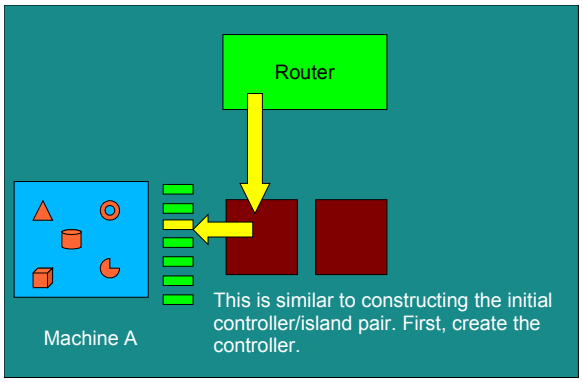
At this point, we can create a new Island from scratch using the Controller. We can also populate the Island and have the objects inside of it begin the process of sending their own internal messages into the message queue. Once the Island exists, we still need to be allowed to participate in it, which allows us to send it external messages generated by user events. Like joining, this is simply making a request to the Island to be allowed to participate. If granted, our Controller receives a list of facets, which is a kind of encrypted dictionary of messages that we are allowed to send from the user through the Router to the Island.



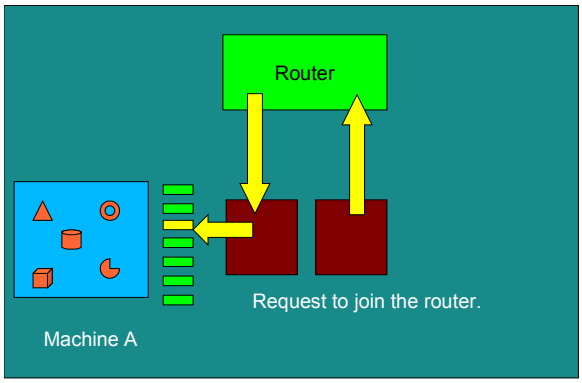
Adding Users

The new users need to be able to join a Croquet session while it is running, with minimal cost to the other users. If done properly, the other users might not even notice that another user has joined the session apart from seeing a new avatar appear in the scene.

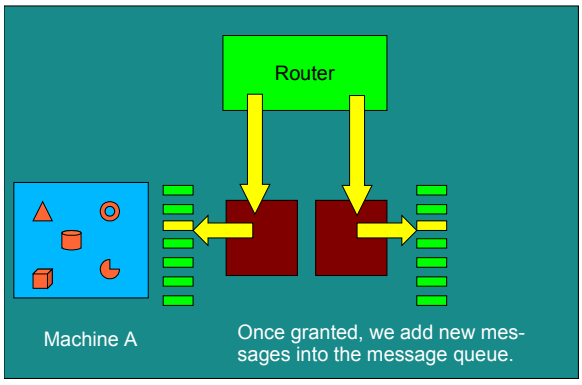
Since a Croquet Router already exists, we only need to create a new Controller on our local machine. This is identical to creating the original Controller on the original users machine.



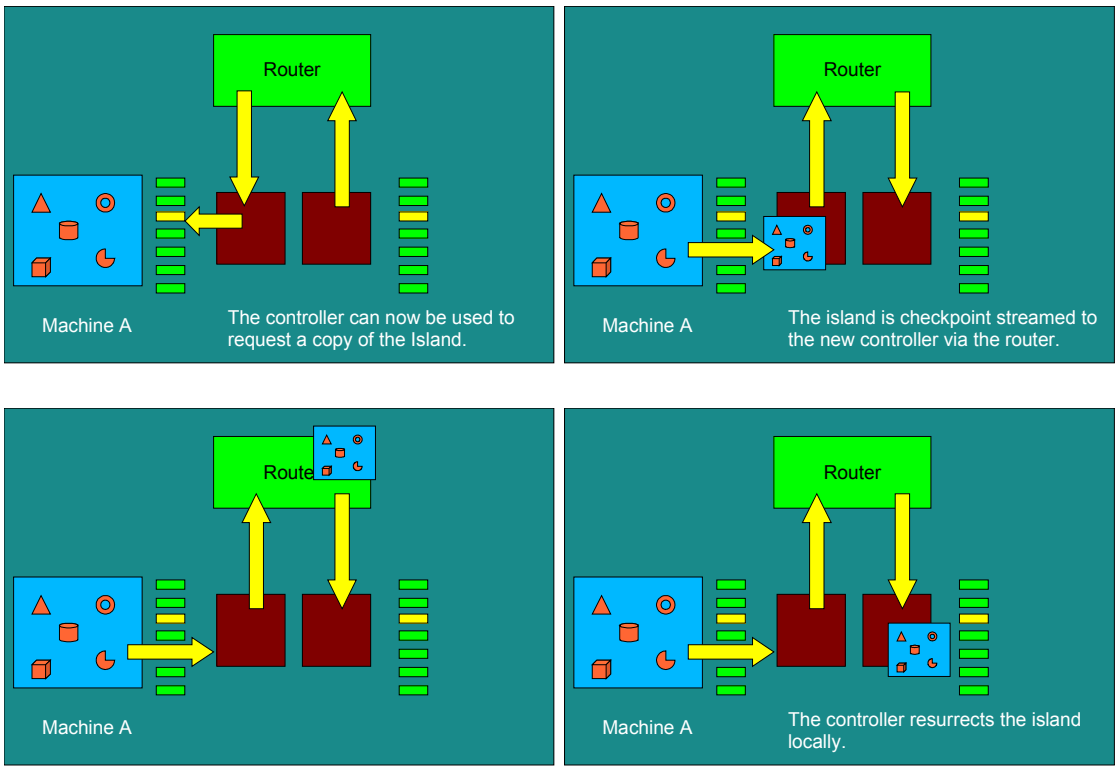
Just as before, the new Controller requests that it be allowed to join the ongoing Croquet session.



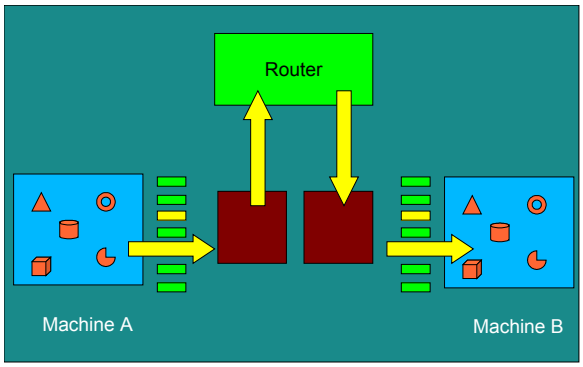
Once permission to join is granted by the Router, the new Controller will begin receiving messages from the Router. In this case, these messages will likely include events generated by the current users of the Croquet Island, so these messages are extremely important.



Now, instead of creating a new local Island, the Controller needs to request a copy of the current Island from the other user. The Router forwards this request to the original Controller and the Island is “checkpointed”, in that a copy is made at a particular instant in time and streamed out to the new user via the Router.



Once the Island has been copied over to the new users machine – the message queue is truncated to the time of the most recent message executed by the Island, and execution seamlessly picks up from that point. The two users are now perfectly synchronized with identical Islands.



Participating

At this point, the new user has basically read-only access to the Island and must also ask to be allowed to participate. Again, it requests this from the Router and if granted the new user can begin to interact with his peers inside of the Island. The current system allows for both joining and participating with one message via the Croquet Harness.

Nice Side Effects

Because no messages are ever lost, and because the original message senders cannot specify when a message is to be executed, latency does not create timing or synchronization problems, just feedback problems. Systems will act sluggish if you have a higher latency, but the contents of the Island will remain identical between all users regardless.

This also means that users are not punished for having a high-latency participant sharing an Island, though the high-latency participant may have a less than satisfactory experience.

Since Routers are independent of Island/Controller pairs, they can be positioned anywhere on the network. This means that they can be moved to a position of minimal group latency, or onto centralized balanced latency servers. Routers can even be moved around if necessary to improve latency for specific users or groups for a certain time period.

The Future of Croquet Objects

The real work of Croquet is actually performed by the objects that are inside of the Islands. These are the objects that know how to display themselves, respond to external user events, and perform time-based simulations. They can be 3D objects that get rendered using OpenGL, or 2D objects that lie flat on the screen, or even zero-D objects that have no visual representation at all, but can perform complex computations.

In fact, there are really no special "Croquet" objects. The real distinction is that objects inside of Islands can send and receive future messages. These are virtually any message that an object understands, but sent into the future to be executed at an explicit later time. The syntax is basically the same as sending a normal message to an object, except we need to specify how far into the future the message will be executed.

As an example, if we want to rotate a cube in the 3D world by ten degrees around its y-axis we would normally write:

```
cube addRotationAroundY:10.
```

This would be executed immediately, and the cube would be rotated 10 degrees. If instead, we wanted to perform this operation sometime in the future, perhaps one second, we would write something like this:

```
(cube future:1000) addRotationAroundY:10.
```

The only real difference is the #future:1000, that specifies we want the next message - #addRotationAroundY: - to be executed 1000 milliseconds, or one second from now.

Virtually any object can be sent messages this way. Outside of an Island, we only have indirect references to the objects inside of it. We can still send messages to these inside objects via the reference, but we cannot specify when these messages are actually executed. To send a message to the cube inside of the Island, we first need to have a TFarRef to the cube – call it farCube, and to send the translation message we would do something like the following:

```
farCube future addRotationAroundY:10.
```

We cannot specify how far into the future this message will be executed. The only guarantee is that the Router will attempt to have it execute as soon as possible. If it is necessary to have a delayed future send, then you will need to write another method that in turn performs a future send of the required time. As an example, if you want to have the rotation triggered in five seconds after the external future send, you could write the following method:

```
MyCube>>#addRotationAroundY: angle when: time

        (self future: time) addRotationAroundY: angle.
```

Then, you would execute the following from outside of the Island:

```
farCube future addRotationAroundY:10 when: 5000
```

Note that this is still relatively undefined. All you know is that the actual add rotation message will be executed exactly five seconds after this message is executed, and all you know about that is it will be dependent on the best efforts of the Router and network.

Programming Croquet – Quick and Dirty

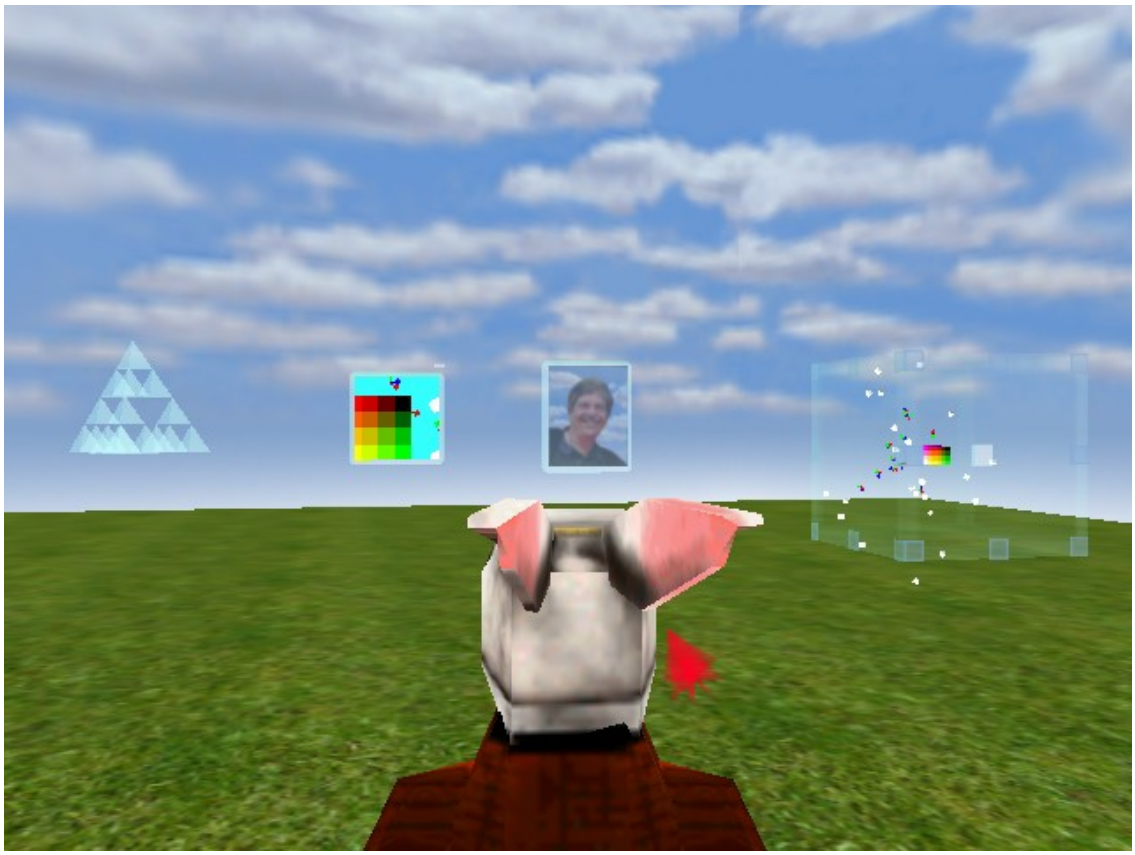
> This section is a deconstructionist point of view – starting with a working system and examining each of the pieces in context.

The quickest way to get your feet wet is to jump into the deep end of the pool. Well, here it is. Rather than try to construct a working system from the ground up, I will start with a working system and take it apart bit by bit. You can follow along with the current Croquet release. Once you have gained something of a global understanding of the pieces that make up a working Island, we will dive down into each of the pieces and describe how they work and how to make new ones.

So let's start with the code. The classes we will be looking at are used to create the Croquet(Master) demo. They are CroquetHarness, SimpleWorld and RecurseWorld. The CroquetHarness class is the primary interface between the user and the Croquet worlds. We will be using it to aid construction of the two Islands that we will be using. SimpleWorld and RecurseWorld are two simple classes that are used only to generate two simple Islands that are linked together using portals.

> Of course you understand that the SimpleWorld and RecurseWorld are nothing more than scripts masquerading as classes. The important thing is what they do – not what they are. We are working on more flexible scripting mechanisms.

As simple as these worlds are, they illustrate many of the things you will need to know when constructing your own objects and islands. Let's start with the SimpleWorld class, which defines that space.



The image above is of the resulting world available when you activate the Croquet(Master). We are currently inside of the SimpleWorld Island. At the center of the screen is the white rabbit avatar – your representative inside this world. The objects inside of the SimpleWorld are, from right to left, a Sierpinski pyramid, a regular 2D portal to the RecurseWorld, a picture of Alan Kay that is more than it seems, and a 3D portal into the RecurseWorld again. There is a small 3D colored cube at the center of the 3D portal that is the same cube as is visible through the 2D portal.

SimpleWorld

The Simple World was designed to be an easy space to construct and explain. There are only a few objects in the space, but they are all pretty interesting. In particular, there are two kinds of portals inside of SimpleWorld, which is one of the key ideas behind Croquet – virtually linked spaces that work in the same way that web links work. We are more interested in how we arrange this particular world than how each of the objects works at this point, so let's look at the initialize script inside of SimpleWorld. Currently, we use the class SimpleWorld as a means to construct the space, but once the construction is complete, we have no need of the SimpleWorld object, which is essentially empty. We are using the #initialize method as a script.

SimpleWorld>>#initialize

Remember that the #initialize method here is being used as a script to construct these spaces. The actual SimpleWorld object is of no real use to us and is thrown away as soon as it is done constructing the contents of the new Island.

```
0.    SimpleWorld>>#initialize

      | space portal sky win p3 |

1.    space := TSpace new.
      space registerGlobal:#masterSpace.

2.    self makeLights: space.
      self makeFloor:space fileName:'lawn.bmp'.
      self makeKay: space.
      self makePyramid: space.

3.    win := TWindow new.
      win translation: 0@0@-20.
      space addChild: win.

4.    portal := TPortal new.
      portal registerGlobal:#portal1.
      portal extent:5@5.
      win contents: portal.

5.    sky := TSkyBox new initializeWithFileName: 'JUL'.
      space addChild: sky.
      sky step. "get going"

6.    p3 := TPortal3D new.
      p3 translation: 10@-1@10.
      space addChild: p3.
      p3 registerGlobal:# portal3D.

7.    TScrollBox3D new initializeWithContents: p3.
```

We have numbered the “paragraphs” of the code for reference below. Virtually all of the objects that we will be dealing with in this construction are

subclasses of TFrame, which is the base class of all of the 3D graphics objects in Croquet. Subclasses of TFrame use the pattern of adding a "T" to the name, hence TSpace, TLight, TCube, TWindow…

0. Starting from the top, though this is an instance method for SimpleWorld, it could just as easily be a class method, or later on a script. In this case, it is just the instance #initialize method that will get called when the SimpleWorld is constructed by the Island we are placing this content into. Later in this section, we will show you how we construct the Island and how we hand it these construction classes.

1. A TSpace is constructed. A TSpace is simply the root object in which all other objects in a particular scene are ultimately contained. That is, all of the objects in the image above – the white rabbit avatar, the windows, the pyramid, and the 3D portal are children of the TSpace and in turn the TSpace is their parent. This is really a tree structure, where each node of the tree is a kind of TFrame, including the TSpace, and where the TSpace is the root of the tree. Once the TSpace is created, we need to define a global name for it so that it can be referenced from outside of the Island. In this case, we register the new TSpace with a global name of #masterSpace by calling: `space registerGlobal:#masterSpace`, because this will be the main space that we use for this example.

2. Once we have constructed the TSpace, we begin the process of generating the content for it. The very first thing we need in a new space is the ability to see what is inside of it, and this requires that we light the scene. In Croquet, a TLight is just another kind of TFrame and can be positioned inside of a TSpace in the same way. In this case, we already have a method that sets up a default set of lights for us that we can use by calling: `self makeLights: space`. We also have a method that we can call that will construct a simple floor for us to stand on, where we can pass on which texture we want it to use. The floor is constructed with: `self makeFloor: space filename: 'lawn.bmp'`. In the same way, we construct the TWindow containing Alan Kay's picture using: `self makeKay: space`. and the Sierpinski pyramid using `self makePyramid: space`. We will dive into these methods a bit later.

3. We will be creating a TPortal to link this space to another one later on, but we also want to place this TPortal into a TWindow that is an object that, like windows in 2D, will allow us to move the its contents around around and resize it. Constructing a TWindow is pretty much the same as a TSpace, it is: `win:= TWindow new`. To move the new window to the proper location in front of us when we start up, we send it: *win translation:0@0@-20* , which positions it 20 units in front of the avatars

start position which is at 0@0@0. The pattern “1@2@3” defines a three element vector with 1 = x, 2 = y, and 3 = z. Following common practice for graphics systems, the positive x-axis is to the right as you look at an object, the positive y-axis is up, and the positive z-axis is the "right hand" cross product (e.g., back towards the observer), Finally, we place the new window into the space’s hierarchy as a child with: space addChild: win.

4. Now we construct a TPortal, which is a kind of link between spaces and will be used to view and access the second space we will be constructing. Again, much like before, we construct a new TPortal with: portal := TPortal new. Since we do not have the references to the second space yet, we need to set up access to this TPortal in the same way we did for the TSpace before using : portal registerGlobal: #portal1. This gives us a handle later so that we can fill in the #linkTo: of this new portal. Next, we need to specify the initial size or extent of the TPortal and finally, we add it to the TWindow as content: win contents: portal. We do not add the new TPortal as a simple child of the TWindow because the TWindow will be taking an active interest in the new TPortal as the user manipulates the TWindow. Simply adding the TPortal using #addChild: would not give the TWindow any information about how the object is intended to be handled, or whether it even should be. It may well be just an ornament of some kind placed on the TWindow.

5. Now we need to add the blue sky above and the green grass below us. We construct a TSkyBox with a slightly different pattern from before. We also need to add some information about what sky we wish to use. In this line, we both construct the base sky box and then specify part of the texture name: sky := TSkyBox new initializeWithFileName:’JUL’. Once we have the new sky object, we add it to the space in the same way we added the TWindow before with: space addChild: sky. This places the sky as another child of the original TSpace. Next, we want the sky to appear as if the clouds are moving, which is done by simply slowly rotating the skybox around the observer. We start the process of by sending the first #step message: sky step.

6. There are two kinds of portals in Croquet – the 2D type – which act as doorways or windows into another space, and the 3D portals, which are also known as microworlds. Again, we see the same pattern for constructing this new object: p3:= TPortal3D new. And again, we position our new 3D portal using #translation: and add it to the space as another child using #addChild:. Finally, just as we needed an external Island reference to the TPortal so that we could set its links, we need the same with the TPortal3D: p3 registerGlobal:#portal3D.

7. Finally, we want a way to move the contents of the new TPortal3D around, so we add a 3D scroll box that has handles that allow us to translate the contents and rotate the box itself. We can also scale the contents of the scroll box. We construct this 3D scroll box in a slightly different way from how we constructed TWindows. Here we combine in one line the construction of the TScrollBox3D and installing the TPortal3D contents: TScrollBox3D new initializeWithContents: p3. You may have observed that this is different from how we set up TWindows, in that we did not add the TScrollBox3D to the TSpace as a child and we did not set its #translation:. The reason is that the TScrollBox3D is designed to install itself around the contents in such a way that the contents don't move around, but the box defined by the TScrollBox3D is located in the correct place. The TScrollBox3D removes the contents from the TSpace, adds itself to the TSpace, and then adds the contents in turn as a child to itself. Don't worry about how all of this works, you will be seeing plenty of examples of this kind of thing inside of Croquet.

Now let's take a look at how we added some of the objects that make up this world. Later on, we will dive into how we actually designed these objects to do what they do.

SimpleWorld>>#makeLights:

The very first thing we added to the new space in SimpleWorld were the lights. The code that does this is:

```
0.      SimpleWorld>>#makeLights: space

        | light |

1.      "warm light in northwest corner."
                light := TLight new.
2.              light ambientColor: #(0.48 0.35 0.4 0.5).
                light diffuseColor: #(0.48 0.35 0.4 0.6).
                light specularColor: #(0.3 0.23 0.2 0.5).
3.              light rotationAroundY: -130.
                light addRotationAroundX: 60.
4.              space addChild: light.

        "cool light diagonally opposite."
                light := TLight new.
                light ambientColor: #(0.4 0.35 0.48 0.5).
                light diffuseColor: #(0.4 0.35 0.48 0.6).
                light specularColor: #(0.2 0.23 0.3 0.5).
                light rotationAroundY: 132.
```

```
            light addRotationAroundX: 60.
            space addChild: light.

        " fainter green light in third corner."
            light := TLight new.
            light ambientColor: #(0.1 0.2 0.1 0.3).
            light diffuseColor: #(0.1 0.23 0.1 0.4).
            light specularColor: #(0.3 0.2 0.3 0.3).
            light rotationAroundY: 42.
            light addRotationAroundX: 60.
            space addChild: light.

        "and another opposite it"
            light := TLight new.
            light ambientColor: #(0.1 0.2 0.1 0.3).
            light diffuseColor: #(0.1 0.23 0.1 0.4).
            light specularColor: #(0.4 0.3 0.2 0.2).
            light rotationAroundY: -42.
            light addRotationAroundX: 60.
            space addChild: light.
```

As you can see, we are adding four new lights. Let's examine just one of them, as they all share the same basic model for setting up.

1. We create a new light just as we create any new object, in this case it is: `light := TLight new`. This creates a default directional light. A directional light is a light where all of its rays are parallel to the direction of the light, simulating a light source that is extremely far away, like the sun. We can also set this light to be a point light or a spotlight.

2. Once we have created the light, we need to set the intensity of its colors. There are three important parts to a light in Croquet, the ambient color, the diffuse color, and the specular color. Each of these is specified using a standard red, green, blue, alpha array where each element needs to be a value between 0 and 1. Since TLights ultimately resolve to be OpenGL lights, these values correspond directly to the values used to specify OpenGL lighting.

3. Next we specify the orientation of the light, and since this is a directional light, the orientation is particularly important. The default direction for the light is going straight up the y-axis. We modify its orientation by specifying that we rotate around the y-axis (the up down axis) first: `light rotationAroundY: -130`. Then we rotate around the x-

axis, which pitches the light down: light addRotationAroundX: 60. We use #addRotationAroundX: instead of #rotationAroundX: because we have already modified the rotation of the light by rotating it around the y-axis. If we just set the rotation around the x-axis, instead of adding it, we would actually have an undefined transformation, and probably nothing like what we really want.

4. Remember that the TLight is just another TFrame, so add it to the space in the same way we added all of the other TFrame subclass objects: space addChild: light. We have now added lights to our space that will give us something to look at, and give a bit more dimensionality to the scene.

SimpleWorld>>makeFloor:fileName:

The next thing we added was the grassy floor. This is also a very simple method.

```
0.    SimpleWorld>>makeFloor: space fileName: txtrName

      | stone txtr |

1.    txtr :=  TTexture new initializeWithFileName: txtrName
                              mipmap: true
                              shrinkFit: false.
      txtr uvScale: 8.0@8.0.

2.    floor := TCube new.
      floor extentX:80 y:0.5 z: 80.
      floor translationX: 0 y: -6.0 z: 0.0.
      floor texture: txtr.
      floor objectName: 'floor' copy.
      space addChild: floor.
```

1. There are two new objects that we create inside of #makeFloor:fileName:. The first is a TTexture. This is a kind of bitmap that gets wrapped around a 3D object. We should note that TTextures are also subclasses of TFrame, which means we can drop them directly into a scene themselves. This allows us to see a TTexture directly without it being wrapped around another object. Creating a TTexture takes a few more parameters than

usual: txtr := TTexture new initailizeWithFileName: txtrName mipmap: true shrinkFit: false. The txtrName in this case is the name of the bitmap file (bmp, jpg, gif, or png) that you want to load. Mipmap indicates whether or not we generate smaller versions of the texture for getting a better quality visual when we are further away, and shrinkFit is used to define whether we shrink the texture to the next smallest power of two or grow it to the next largest. We need to do this because many systems cannot handle textures that are not scaled to be some power of 2 (2,4, 8, 16, 32, 64, …). The next line defines how often the texture is replicated over a surface. In this case, we have set it to be 8x8 when stretched over the top. If you look closely at the floor of the SimpleWorld, you may notice that the texture is in fact repeated.

2. Once we have the texture, we need to build a 3D object that will be the actual floor. We will use one of the simplest objects we have, which is a TCube. We construct the TCube just as everything before: floor := TCube new. We need to make the floor pretty big to hold everything even in our little space. We set the size of the TCube with: floor extentX:80 y: 0.5 z:80. This makes a cube with that is 80x80 in the main plane and ½ a unit high. Then we position it, primarily drop it 6 units down in the y direction: floor translationX:0 y:-6 z:0. Note that this is a slightly different variant of #translation: than what we used before. They have identical end results, and you can use them interchangeably. Next we add the new TTexture object to be mapped onto the TCube: floor texture: txtr. We also give the floor a name, just in case we ever need to find it again: floor objectName: 'floor' copy. We make a copy of the string 'floor' to ensure that only this copy is actually inside the Island. Last of all, just as we have done some many times before, we add our new floor to the space: space addChild: floor. Now we are ready to walk on it.

The other two methods: #makeKay: and #makePyramid: are similar to the examples above and worth taking a look at.

RecurseWorld

Now that we have created our first simple world, let's look at the world that both of our portals are linked to. After that, let's look at how we set these worlds up and link them together.

The RecurseWorld is also a "simple" world in that it is also extremely easy to set up. It has a single portal in it that will be linked back to the SimpleWorld. It

also has a number of other interesting objects that we will take a look at. The #initialize script for RecurseWorld is a bit longer, but it isn't hard to understand at all. This is just another example of a kind of Croquet space we can build, and you will no doubt see the significant degree of similarity between the RecurseWorld and the SimpleWorld described above.

```
0.    RecurseWorld>>#initialize

      | frame space cube cube2 portal win number count |

1.    space := TSpace new.
      space color: (VectorColor r:0.2 g:1.0 b:1.0 a:1.0).
      space registerGlobal:#recurseSpace.

2.    TLight makeLights: space.
      cube := TCube new.
      space addChild: cube.

3.    50 timesRepeat:[
            cube2 := TCube new.
            frame := TDemoFish new.
            frame target: cube.
            frame contents: cube2.
            frame translation:
                  ((20-(40 atRandom))@
                  (20-(40 atRandom))@
                  (20-(40 atRandom))).
            space addChild: frame.
            frame addRotationAroundX: 360 atRandom.
            frame addRotationAroundY: 360 atRandom.
            frame addRotationAroundZ: 360 atRandom.
      ].

4.    25 timesRepeat:[
            frame := TSpinTest new.
            space addChild: frame.
            frame translation:
                  ((10-(20 atRandom))@
                  (10-(20 atRandom))@
                  (10-(20 atRandom))).
            frame addRotationAroundX: 360 atRandom.
            frame addRotationAroundY: 360 atRandom.
            frame addRotationAroundZ: 360 atRandom.
      ].
```

```
5.  number := 4.
    count := 1.
    1 to: number do:[:i |
        1 to: number do:[:j |
            1 to: number do:[:k|
                cube := TDragCube new.
                cube translation:
                    ((i-1-(number/2.0))@
                    (j-1-(number/2.0))@
                    (k-1-(number/2.0))).
                cube setColor:
                (VectorColor r: number-i/number asFloat
                g:number-j/number asFloat
                b:number-k/number asFloat a:1.0).
                cube objectName: count.
                count := count+1.
                space addChild: cube.
            ].
        ].
    ].


6.  win := TWindow new.
    win translation: 0@0@15.
    win rotationAroundY:180.
    space addChild: win.

7.  portal := TPortal new.
    portal translation: 0@0@0.
    portal registerGlobal:#portal1.
    portal extent:5@5.
    win contents: portal.
```

1. Starting from the top, we see the standard creation and registration of a TSpace. The big difference this time is that we setting a color for the TSpace: space color: (VectorColor r:0.2 g:1.0 b:1.0 a:1.0). This is the green background color you see when you look through the TPortal. You can't see the color of the SimpleWorld, primarily because it is blocked by the TSkyBox.

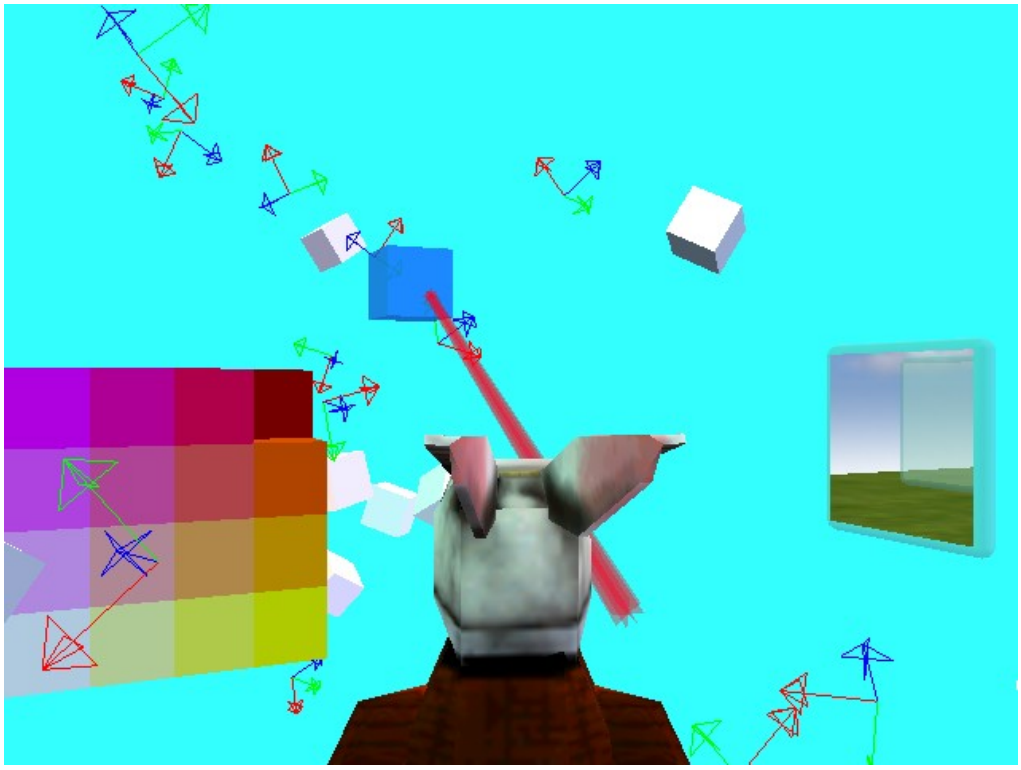
2. Just as before, we need to add some lights. The difference is that this time, we are using a class method to set up the lights for us: TLight

makeLights: space. This is actually the same end result, but is available to any method that wants to generate these kinds of lights.

3. Here we are going to construct 50 new small cubes and set them swimming around inside of our new space. We first generate a TCube as we have done so many times before, then we generate a TDemoFish, which is simply a kind of TFrame that moves around the space in a semi-random way. We add the new TCube to the TDemoFish object and the result is that it makes the TCubes move around the world and respond to some degree to the users pointer.

4. Here we are creating 25 very simple spinning objects. Basically a TSpinTest is nothing more than a TFrame that knows how to spin itself around. In fact, if we look closely at the TSpinTest initialize, all it does is call: super initialize, which is used to construct the base TFrame and begin the process of spinning by calling: (self future: (250 atRandom+250)) step, which sets the object spinning at an update rate chosen at random between 250 and 500 milliseconds. The three colored arrows that are rendered by the TSpinTest are the default rendering of TFrame.

5. Next we construct a larger cube made up of a number of smaller colored cubes. Each of these cubes is a TDragCube and allows the user to manipulate it by selecting any of the cubes surfaces and dragging parallel to that surface. Notice here that we have moved the red corner cube away from the rest of the cubes inside of the new space. It is currently selected, so it actually displays as blue instead of red, but once we release it, it will be red again. You can also see the white cubes that are swimming around in the space as part of the TDemoFish and the perpendicular arrows that make up the TSpinTest. These arrows represent the local x, y, and z-axis of the object being displayed and is the default rendering method for TFrames.



6. The next step is simply to construct a TWindow as we did before to hold the portal that will point back to the original space that we constructed. Note that this time, we are rotating the TWindow 180 degrees. We don't actually have to do this, as the TPortals work fine no matter what direction they face. This is just a way to direct the portal to where the interesting stuff is inside of the space.

7. Finally, we create the TPortal that will be added to the TWindow and register it just as before. You can see the SimpleWorld through the window at the right in the image above.

We have just completed two new spaces and set up the links that will be made between them. We have not actually made those links yet for a number of reasons. First, there is no way for one Island to reference the contents of another. Islands are self-contained objects and can only access the other objects that are inside of the Island. We can register the objects inside of the Island as we did so that we can access them outside using a TFarRef, but an Island still can't even have direct access to that. Instead we need to hook these portals up between Island while we are actually outside the Island and we need to use an object that provides only a simple reference to the contents of another Island called a TPostcard.

Setup Master

Defining the contents of an Island is a critical first step, but now we need to construct these new Islands and link them together using their portals. To do this, we have written a method inside of CroquetHarness called #setupMaster. This method can actually exist anywhere, including in another object, but we are including it here for now. At some later point, this too will be an external script.

```
0.      CroquetHarness>>#setupMaster

        | space sync island portal spc pc rIsland rportal rpc portal3D
rspace rspc |

1.      island := self createIsland: SimpleWorld named: 'Master'.
2.      space := island future at: #masterSpace.
        portal := island future at:#portal1.
        portal3D := island future at:#portal3D.

3.      spc := space future postcard.
        pc := portal future postcard.

4.      spc whenResolved:[
                self addIsland: island postcard: spc value.
                sync := viewPortal future postcardLink: spc value.
                sync whenResolved:[
                        doRender := true]. "ready to render"
        ].

5.      rIsland := self createIsland: RecurseWorld named: 'Recurse'.
        rspace := rIsland future at: #recurseSpace.
        rspc := rspace future postcard.
        rportal := rIsland future at: #portal1.
        rpc := rportal future postcard.

6.      pc whenResolved:[
                rpc whenResolved:[
                        self addIsland: rIsland postcard: rpc value.
                        portal future postcardLink: rpc value.
                        rportal future postcardLink: pc value.
                ].
        ].
```

```
7.      rspc whenResolved:[
                portal3D future postcardLink: rspc value.
        ].
```

1. The CroquetHarness has a number of helper methods and #createIsland:named: is one of the more useful ones. Basically, this method sends the #initialize method to the object generated by the class in such a way that the results of the #initialize are all inside of the new Island. When we construct the new Island with: island := self createIsland: SimpleWorld named: 'Master', the arguments are the SimpleWorld class and the name of the new Island that we will be using to find it again. The 'Master' Island is the world we find ourselves in by default when Croquet starts up. Also notice that "island" is not a direct reference to the new Island object, but is actually a TFarRef.

2. As you will recall, when we created the SimpleWorld Island, we created global references to a number of objects that inside of it. Here we are dereferencing these globals so that we can access them in some way from outside of the Island. The space, portal, and portal3D objects are actually TFarRefs. These are a kind of handle to the original object that allows us to send future replicated messages to the original object.

3. Any TFrame inside of an Island can generate a TPostcard, which is very similar to a URL link in a web page. It contains such information as which Island the TFrame is in, the ip address and port number, and the object ID and name. It has no direct references to any internal Island object, so it is safe to save in other contexts. In fact, it is intended to be used by other Islands to link back to the original Island. Here we are generating two TPostcards for the same Island – a TPostcard pointing to the entire TSpace which will be used by the TPortal3D, and a TPostcard pointing to the TPortal that is inside of the TWindow. Actually, when we send future messages that return values in this way, we don't actually retrieve the original object, but instead get a TPromise, which is basically a reference to an object that will at some point later have the actual requested object. This is because the #future message is not being executed immediately, but will be executed as soon as possible by the Router. In this case, the variables spc and pc are promises and not the actual TPostcards we need.

4. Since we don't yet have the TPostcard for the requested TSpace, we have to wait a bit until this field in the new TPromise is resolved. To do that we use: spc whenResolved:[….]. Once we have this, we can get

the actual TPostcard by sending #value to the TPromise, hence the TPostcard for the main TSpace inside of the SimpleWorld Island can be had with: spc value. Once we have a TPostcard to this Island, we need to register the Island with the CroquetHarness. Any TPostcard will work for this: self addIsland:island postcard: spc value. The viewPortal in the next line is actually the main viewer for local user of Croquet. It is a TPortal that is aligned with the view screen. We need to link the space inside of our new Island to this viewPortal so that we can see and interact with the contents of this new Island. To do that, we drop the TPostcard associated with the new Island's TSpace, spc value, as the #postcardLink: of this portal: sync:= viewPortal future postcardLink: spc value. The return value "sync" is another promise that we use to determine when the TPortal is ready to begin rendering the new space via the new TPostcard.

5. We do essentially the same thing with RecurseWorld. We create the Island and grab its global references and again generate TPostcards.

6. Now we have postcards to TPortals from both the SimpleWorld and the RecurseWorld Islands. What we are doing here is first, adding the RecurseWorld Island to the CroquetHarness, then we are setting the TPortal in the SimpleWorld to point to the TPortal in the RecurseWorld and the TPortal in the RecurseWorld to point back to the TPortal in the SimpleWorld.

7. Finally, we are setting the TPortal3D TPostcard to be the link to the entire RecurseWorld space.

Again, this method can exist anywhere, all you need is a reference to the CroquetHarness and the names of the Island constructer classes.

A Croquet Object

Of course, Islands are made up of a collection of objects that know how to display themselves to the users, know how to react to user events, and know how to express behaviors over time. We have already been introduced to a number of objects when we created the SimpleWorld and the RecurseWorld above. Let's dive in even deeper and examine one of these Croquet objects.

The object we are interested in is the TDragCube, which is a simple subclass of TCube. The TCube is basically a 6-sided parallelogram, not necessarily a cube, though that is its default. The TCube only knows how to render itself in 3D. It has no behaviors and cannot handle events on its own. The major addition that TDragCube brings is that it adds event handling to the base TCube class that

allows the user to drag the TDragCube around by any of its sides (hence the name). We will first look at TCube and then at how we extend TCube with TDragCube.

A TCube is one of the simplest objects in Croquet. It is a subclass of TPrimitive, which is a class that provides management of a number of visual properties like materials and transparency. A TPrimitive is in turn a subclass of TFrame, which is the base class of all 3D rendered objects in Croquet. The interesting aspects of a TCube are in how it initializes it data and how it renders it.

TCube>>#initialize

The TCube #initialize method provides us with a default 1x1x1 unit cube as well as setting up additional support objects.

```
0.  TCube>>#initialize
1.      super initialize.
2.      extent :=Vector3 x: 1.0 y: 1.0 z: 1.0.
3.      location := Vector3 x: 0.0 y:0.0 z:0.0.
4.      self initBounds.
5.      changed := true.
6.      self update.
```

All #initialize methods of an object are called implicitly by the class>#new method. This means we do not have to call it ourselves, it is already taken care of.

1. Since a TCube is ultimately a subclass of TFrame, we need to ensure that the TFrame initialize is taken care of, so we have to call it explicitly inside of the TCube initialize.

2. We need to specify the default size, or extent, of the cube, which in this case is 1x1x1.

3. We set the default location to be 0.0@0.0@0.0. This location is different from the object's translation, and is actually added to that translation when rendering.

4. We use a bounding sphere as a first approximation to the object. This sphere is used both to determine when we can cull the object from the scene when rendering it and to determine if the users event pointer might intersect with the object. We need to compute this bound sphere and we do it in the TCube>>#initBounds as below:

```
TCube>>initBounds

	boundSphere := TBoundSphere localPosition: location
		radius: (extent length)/2.
	boundSphere frame: self.
```

The boundSphere is constructed using the TBoundSphere class with arguments of location – the same location described above – an the radius of the resulting sphere, which in this case will be the same as from the center of the box to any one of the corners. Finally, we set: boundSphere frame: self to have the new boundSphere point back to the original TFrame object.

5. We set the changed variable to be true to indicate that we have modified the shape of the TCube.

6. Finally, we send: self update to construct the structure of the TCube to prepare it to be rendered. The #update method, though somewhat long, does nothing more than constructs the vertices, faces, normals and texture coordinates required to render the new TCube.

```
TCube>>#update

	| dx dy dz x y z |

	dx := extent x/2.0.
	dy := extent y/2.0.
	dz :=  extent z/2.0.
	x :=  location x.
	y :=  location y.
	z :=  location z.

	vertices := Vector3Array new: 8.
	vertices at: 1 put: (Vector3 x: x+(dx negated) y: y+(dy
negated) z: z+dz).
	vertices at: 2 put: (Vector3 x: x+dx y: y+(dy negated) z: z+dz).
	vertices at: 3 put: (Vector3 x: x+dx y: y+ dy z: z+dz).
	vertices at: 4 put: (Vector3 x: x+(dx negated) y: y+dy z: z+dz).

	dz := dz negated.

	vertices at: 5 put: (Vector3 x: x+(dx negated) y: y+(dy
negated) z: z+dz).
	vertices at: 6 put: (Vector3 x: x+dx y: y+(dy negated) z: z+dz).
```

```
        vertices at: 7 put: (Vector3 x: x+dx y: y+ dy z: z+dz).
        vertices at: 8 put: (Vector3 x: x+(dx negated) y: y+dy z: z+dz).

        normals := Vector3Array new: 6.
        normals at: 1 put: (Vector3 x: 0.0 y: 0.0 z: 1.0).
        normals at: 2 put: (Vector3 x: 0.0 y: 0.0 z: -1.0).
        normals at: 3 put: (Vector3 x: 1.0 y: 0.0 z: 0.0).
        normals at: 4 put: (Vector3 x: -1.0 y: 0.0 z: 0.0).
        normals at: 5 put: (Vector3 x: 0.0 y: 1.0 z: 0.0).
        normals at: 6 put: (Vector3 x: 0.0 y: -1.0 z: 0.0).

        quadFaces := #(1 2 3 4   8 7 6 5   2 6 7 3   1 4 8 5   4 3 7 8   1 5
6 2) asIntegerArray.
        txtCoords := Vector2Array new: 4.
        txtCoords at: 1 put: (Vector2 x: 1.0 y: 0.0).
        txtCoords at: 2 put: (Vector2 x: 0.0 y: 0.0).
        txtCoords at: 3 put: (Vector2 x: 0.0 y: 1.0).
        txtCoords at: 4 put: (Vector2 x: 1.0 y: 1.0).
        self resetDisplayList.
        changed := false.
```

TCube>>#renderPrimitive:

Once we have initialized our TCube, we can render it to the screen. In the case of a subclass of TPrimitive, as TCube is, we define only one render method, #renderPrimitive. The TPrimitive superclass defines when and how the #renderPrimitive: method is used.

```
renderPrimitive: ogl
        | faceCount uv tc |

        changed
                ifTrue: [self update].
        uv := 1@1.
        self texture ifNotNil:[uv := self texture uvScale.].
        faceCount := 1.

        ogl glBegin: GLQuads.
        normals do:[ :norm |
                ogl glNormal3fv: norm.
                1 to: 4 do:[ :cnt |
                        tc := (txtCoords at: cnt) copy.
                        tc x: (tc x*uv x) y:(tc y*uv y).
                        ogl glTexCoord2fv: tc;
```

```
                    glVertex3fv: (vertices at:
                            (quadFaces at:  faceCount)).
                    faceCount := faceCount+1.].
        ogl glEnd.
```

At this point, the important thing to notice is that #renderPrimitive, as well as all render methods in Croquet, are handed an interface to OpenGL as an argument to the method. The OpenGL methods called here are the same ones that we would call using any other language or interface. Basically, a Croquet object has full access to an extremely powerful graphics capability and has full control of the engine. We won't go into detail about what is going on with this rendering, but there are quite a number of OpenGL resources you can use to help.

TCube>>#pick:

Croquet objects also need to know how the will get picked by the user. That is, when the user attempts to select them with a pointer, the object has the responsibility of determining if the pointer has actually selected it. This is known as a "hit test". The #pick: method is only called if the pointer proves to have already intersected the bounding box, so we already know that the pointer is pretty close to the object. In the case of a TCube, we need to determine if the pointer has intersected with any of the six sides. The code for this is:

```
TCube>>#pick: pointer

        | faceCount |
        faceCount := 1.
        normals do: [ :norm |
                (pointer pickQuad: norm
                        q1: (vertices at: (quadFaces at: faceCount))
                        q2: (vertices at: (quadFaces at: faceCount+1))
                        q3: (vertices at: (quadFaces at: faceCount+2))
                        q4: (vertices at: (quadFaces at: faceCount+3)))
                ifTrue:[^ true].
                faceCount := faceCount+4.].
        ^ false.
```

This is testing each face of the TCube against the pointer one at a time using the TRay>>#pickQuad:q1:q2:q3:q4: method. The TPointer is a subclass of the TRay. There are quite a number of methods in TRay that are there to support this kind of hit test. We will see another one when we examine TDragCube, the subclass of TCube.

TDragCube

The only difference between the TCube and the TDragCube is that the TDragCube can be dragged around by any of its surfaces in any direction that is parallel to that selected surface. Basically, it is a TCube that knows how to handle events.

The first things we need to do is set what color we want the TDragCube to be in its default state. This is pretty simple:

```
TDragCube>>#setColor: bc

	| mat |

	mat := TMaterial new.
	mat color: bc.
	self material: mat.
	baseColor := bc.
```

The argument is a vector color (or at least a four element vector that is coerced to be a vector color). We generate a new TMaterial object, which contains the information about how the surface of our cube will be rendered. We then set the color of the new TMaterial to be the value of the argument and we set the instance variable baseColor equal to the argument as well. You will surely recall in our discussion of the RecurseWorld>>#initialize in section 5 of the code, that we set the color of each of the TDragCubes in that scene using just this method.

The next thing we need to do is to let the Croquet system know that this object is capable of handling events and what those events are. To do that, any object that handles events must provide the #eventMask method. In this case, the TDragCube can handle both pointer over and pointer down events.

```
TDragCube>>#eventMask

	^ EventPointerOver bitOr: EventPointerDown.
```

The only event that TDragCube does not handle is the EventKeyboard. Since we are handling multiple kinds of events, we need to or these event values together using #bitOr:. Each of these events uses a different bit, or location in a byte. The EventKeyboard uses bit 1, so its value is 1 (which is imply the value of the binary number 0001), the EventPointerDown uses bit 2, so its value is 2 (in binary 0010), and the EventPointerOver uses bit 3, so its value is 4 (which is simply the value of the binary number 0100). In this case, we are not using the

keyboard, so we need to compute the bitwise or of EventPointerOver and EventPointerDown which is binary 0110 or 6 in decimal. In the end, all this really means to Croquet is the object will handle any pointer over or pointer down events.

Since the TDragCube can now respond to events when the pointer moves on top of it and leaves it, we need to code these. The first method, TDragCube>>#pointerEnter: event, does nothing more than changes the color of the cube to the standard default TButton overColor that is used throughout Croquet when the user moves the pointer over the TDragCube. This color simply demonstrates to the user that this object can respond to user events.

```
TDragCube>>#pointerEnter: event

        material color: TButton overColor.
        self material: material.
```

If the user moves the pointer away from the TDragCube, then the TDragCube>>#pointerLeave:event method is called, which simply returns the cube to its original base color. You can see this in action by moving your pointer over any TDragCube and then moving it off again. Note that the argument to this method – event, actually a CroquetEvent – is simply left unused. It actually contains a large amount of information about where the cursor is over the object that is extremely useful for more complex interactions, such as the ones described later one.

```
TDragCube>>#pointerLeave: event

        material color: baseColor.
        self material: material.
```

Now things get interesting, because the user can actually start manipulating the TDragCube using the #pointerDown:, #pointerMove: and #pointerUp: events.

```
TDragCube>>#pointerDown: event

        selectedPoint := event selection point.
        norm := event selection normal.
        material color: TButton downColor.
        self material: material.
```

Starting with the #pointerDown: method, we see that it isn't that much more complex than the #pointerEvent: event described above. We now begin to use

the event argument. In this case, we are grabbing the event selection point, which is the value of exactly where on the cube the pointer down event occurred and the event selection normal, which is the normal vector at that same point. These two values will be very important when we start moving the TDragCube around. Finally, just as we set the material color with the #pointerEnter: method, we do the same, but this time we use the default standard color of TButton downColor.

We are now have everything we need to begin dragging the TDragCube around inside of its world. As the user drags their pointer around inside of the space, the TDragCube receives a #pointerMove: message, and the #pointerMove: method can then manipulate the position of the TDragCube to follow the pointer.

```
TDragCube>>#pointerMove: event

        | delta  pointer |
        pointer := event makePointer.
        (pointer frame: self pickPlane: selectedPoint normal: norm)
        ifTrue:[
                delta := selectedPoint - pointer selectedPoint.
                self translation: self translation –   delta.
        ].
```

This method is actually a bit more complex than it might look at first sight. Though we have already determined that the pointer has intersected with this object, we want to actually have the TDragCube follow it around in the plane defined by the selected surface. We have already saved the initial selection point and normal when the #pointerDown: method was called, now we need to use that information along with the changes to figure out how to translate the TDragCube. The first thing we need to do is reconstruct the pointer, as we can't actually hand a TPointer directly to an object. The reason is that the TPointer used here is actually owned by a single user, and is not replicated and can't actually exist inside of an Island. The CroquetEvent event is a replicable object, and can be used to reconstruct the original pointer, which we do here with: pointer := event makePointer.

Once we have the newly reconstructed pointer, we use it to determine if and where the pointer intersects a plane centered at the original selectedPoint with the original normal. This is almost certain to be true, unless the plan is now somehow parallel to the pointer or even behind us. If it is true, then we can compute how much we wish to move the TDragCube in this plane by subtracting the current selectedPoint from the new pointer selectedPoint. Then, we subtract this value from the current translation of the TDragCube.

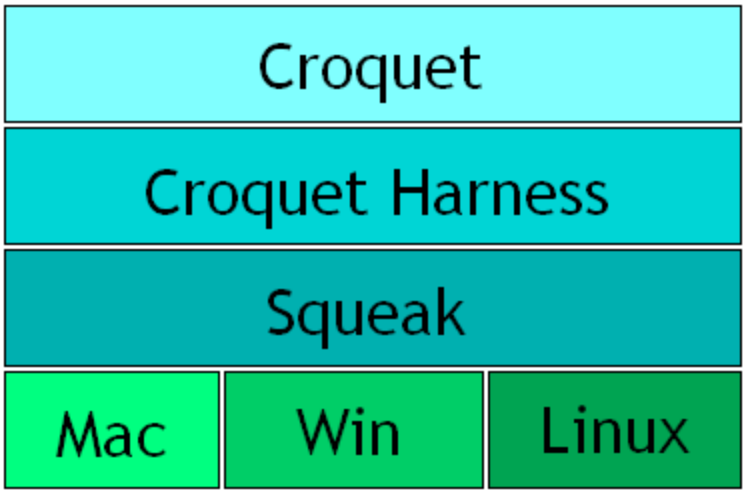
When the user releases the pointer button, the TDragCube is sent the #pointerUp: message. This message is identical to the #pointerLeave: message in that all it does is sets the TDragCube back to its original base color.

```
TDragCube>>#pointerUp: event

        material color: baseColor.
        self material: material.
```

We did not cover the third element of Croquet object, how the object can express behaviors over time, but will do so in a later section.

The Croquet Harness

The Croquet Harness manages the creation of the Croquet Islands, finding and connecting to external replicated Islands, the user's input into Croquet and the rendering output of the system to the users screen. It is the main entry point for creating new Islands and associating the Islands with the user i/o control objects. This allows the Croquet system to be abstracted from the underlying support infrastructure. The intent is that at some future point, all of the UI and presentation layers around Croquet go directly through the Croquet Harness to the underlying hardware. In the mean time, the Croquet Harness is supported on top of both Morphic and Tweak.



The diagram above illustrates how Squeak runs cross-platform across Macintosh, Windows, and Linux, and the Croquet Harness in turn sits on top of Squeak.

Creating New Croquet Objects

Croquet is a component-based architecture that makes it extremely easy and quick for the programmer to create useful collaborative objects. The system was designed from the ground up to make it simple to create truly shared objects of virtually any sort. This section is used to give you an overview of the architecture of these objects, and how the programmer can build his own. We assume a basic understanding of how to program in the Squeak environment and some familiarity with OpenGL. Though there is some overlap between this section and the previous deconstruction section, it is worthwhile to see how we can construct new components from the ground up.

Components

We use the term "component" to describe the basic unit of composition in the Croquet 3D environment. The suite of component level classes is built on top of the TObject base class, which is the fundamental collaboration object. TObject's functionality ensures that when you share an instance object created from a subclass of TObject with other users it maintains a consistent state between them. The TFrame object is derived from TObject and is in turn the base class of all the 3D objects that you can see and interact with in Croquet. The subclasses of TFrame act as frames in an OpenGL rendering hierarchy, as event handlers, and as time-based simulation objects.

At a high level, a component's functionality can be broken up into three main areas:

Graphics – This is how components express themselves visually to the user. The graphics-rendering engine is based on OpenGL.

Events – Events are how the user communicates his desires to the component. This includes mouse and keyboard input.

Simulations – This are how the component expresses itself to the user over time or when stimulated by an event.

Rendering Engine

The philosophy behind Croquet's rendering engine is based on allowing the programmer complete access and control of the underlying graphics library, in this case OpenGL, while at the same time, providing a rich framework within which to embed these extensions with a minimal level of effort. This allows the naïve graphics programmer and 3D artist a way to easily create interesting graphic artifacts with minimal effort, yet allows the expert the ability to add fundamental extensions to the system without the need to modify the underlying architecture.

A rendering frame includes a transform matrix that defines the orientation and position of the object in a 3D space relative to its parent object in the hierarchy as well as the ability to render itself in that position in global space. A rendering message is sent to the object when its position in the hierarchy is reached. The object then calls the appropriate OpenGL library functions to render the object.

Events

An event handler can respond to user events such as keyboard and mouse/pointer events. Again, this interface is quite extensible by the programmer, but the default is that the TCamera carries a TPointer object (a kind of TRay) that tracks the objects that are underneath the current mouse position. A TPointer is a 3D analog to the mouse event object. Instead of being just a 2D position on the screen, the pointer includes vector information, in this case from the camera to the selected object in both global and local (to the selected object) frame transforms.

Keyboard events are also forwarded to the currently selected object when the pointer is over the geometry of the object. This model allows us to embed 2D objects into a scene, where the containing 3D object simply converts the TPointer vector data back into a 2D mouse position on the surface of the 2D object.

Simulations

A component can exhibit behaviors on its own. These can be simple response behaviors based upon it receiving an event, or it can be a time-based change in the fundamental state of the object. This last kind of change we refer to as a simulation. There are three kinds of simulations in Croquet. The first is a simple one that determines the state of an object at render time based only upon the current TeaTime, or a difference between the current TeaTime and a set time in the past, perhaps triggered by an event.

The second kind of simulation is usually used for managing more complex behaviors, those that can't easily be captured by trying to recalculate the state of the world from some time in the past. This is done by literally having the object send a message to itself in the future.

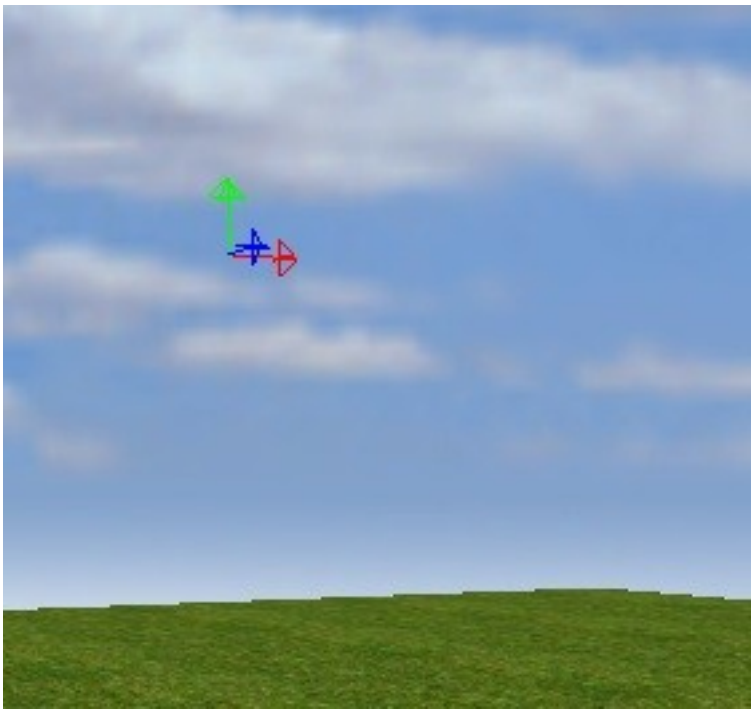
Getting Started: Rendering

To get started in understanding the Croquet architecture, let's just build something. In this case, we will create a simple cube that will be floating above our little 3D world. We are assuming that you already know how to make and edit programs in Squeak. If you don't, we highly encourage you to spend some time getting familiar with it before you dive into this introduction.

The first thing we do is define a new subclass of TFrame. Let's call it TMyCube. There is already a TCube defined that is a bit more efficient and flexible than the one we are about to write. Feel free to check it out.

We can now add a TMyCube object to the scene, even though we have not written any code. To add it to an existing space all you need is:

```
tframe := TMyCube new.
space addChild: tframe.
```



The default rendering method of the base TFrame class simply displays the object's three axes. The code above places the object at the default 0,0,0 position inside of the space. We can easily move it by adding:

```
tframe translation: (1@2@3).
```

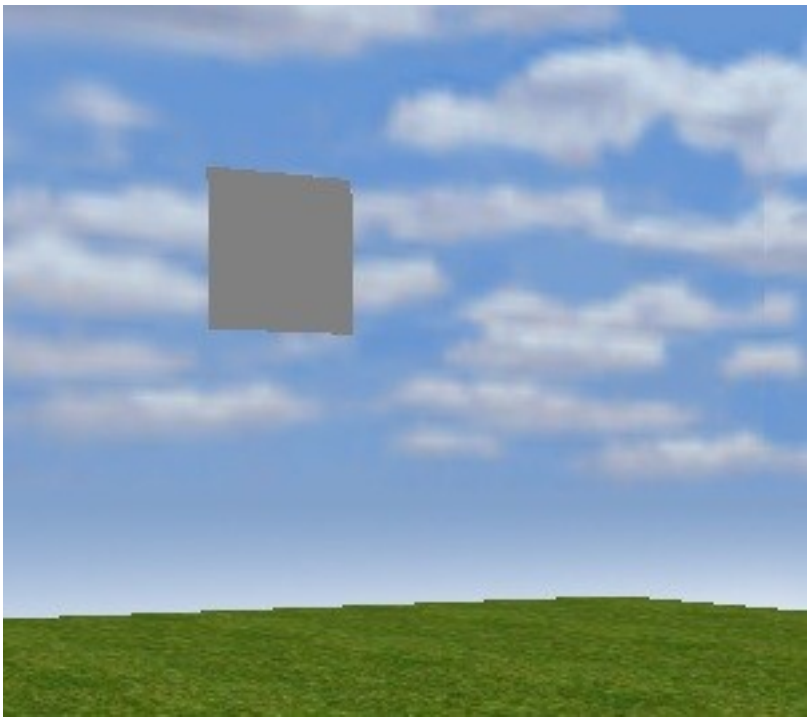
Now let's render one of the cube's faces. Once we do this, we can just replicate the code five more times to do the entire cube.

The Croquet rendering engine is built on OpenGL, but in fact, all of OpenGL is always available to the programmer. Croquet is a good compromise between the ease of use of a retained-mode engine and the flexibility of accessing the low-level graphics routines. In the case of our cube, the base TFrame class takes care of all of the setup and transforms we need to deal with, while the TMyCube class takes care of how to render this particular object.

This is the start of the TMyCube render method:

```
TMyCube>>render: ogl
| dx dy dz |

dx := 1.0.
dy := 1.0.
dz := 1.0.

ogl glBegin: GLQuads.
ogl glVertex3f: dx negated with: dy with: dz.
ogl glVertex3f: dx negated with: dy negated with: dz.
ogl glVertex3f: dx with: dy negated with: dz.
ogl glVertex3f: dx with: dy with: dz.
ogl glEnd.
```

This code generates a 2x2 square one unit away from the center of where the cube will ultimately be. This uses the traditional Smalltalk syntax for accessing OpenGL. The ogl object which is passed in as the parameter to the #render: method is actually the interface to OpenGL and in addition to having methods that give the programmer access to all of the OpenGL library functions, it also maintains some of the global state variables used in rendering.



Croquet extends Squeak syntax to allow you to access OpenGL functions in their 'positional' form. E.g., we support both:

```
ogl glVertex3f: x with: y with: z.
```

as well as:

```
ogl glVertex3f(x, y, z).
```

This allows you to use a representation of the syntax that is nearly identical to that used in the OpenGL textbooks. We can rewrite the #render: method this way:

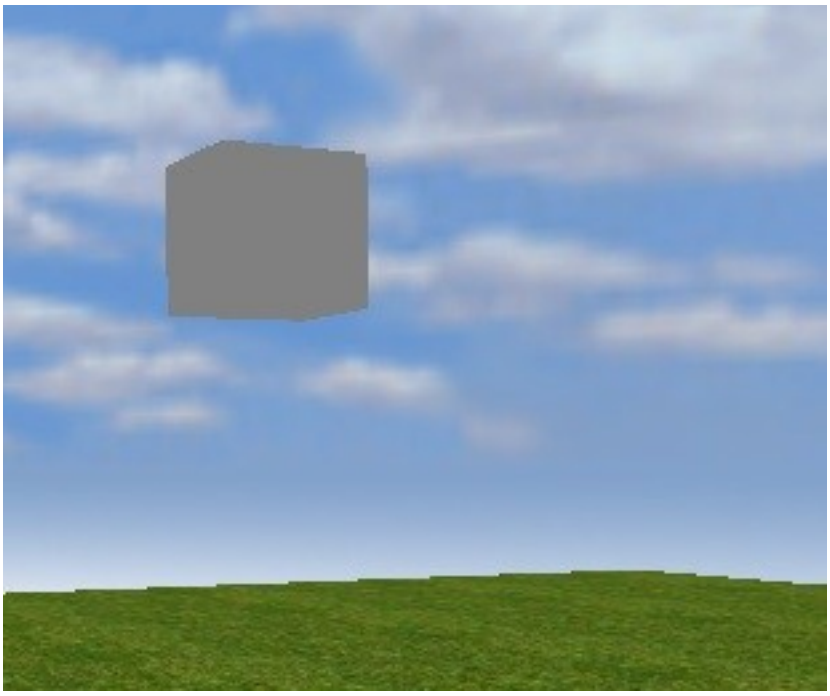
```
TMyCube>>render: ogl
| dx dy dz |

dx := 1.0.
dy := 1.0.
dz := 1.0.
ogl glBegin( GLQuads ).
ogl glVertex3f( dx negated, dy, dz ).
ogl glVertex3f( dx negated, dy negated, dz ).
ogl glVertex3f( dx, dy negated, dz ).
ogl glVertex3f( dx, dy, dz ).
```

```
ogl glEnd.
```

And of course to do all six faces we can just replicate this code with its various permutations:

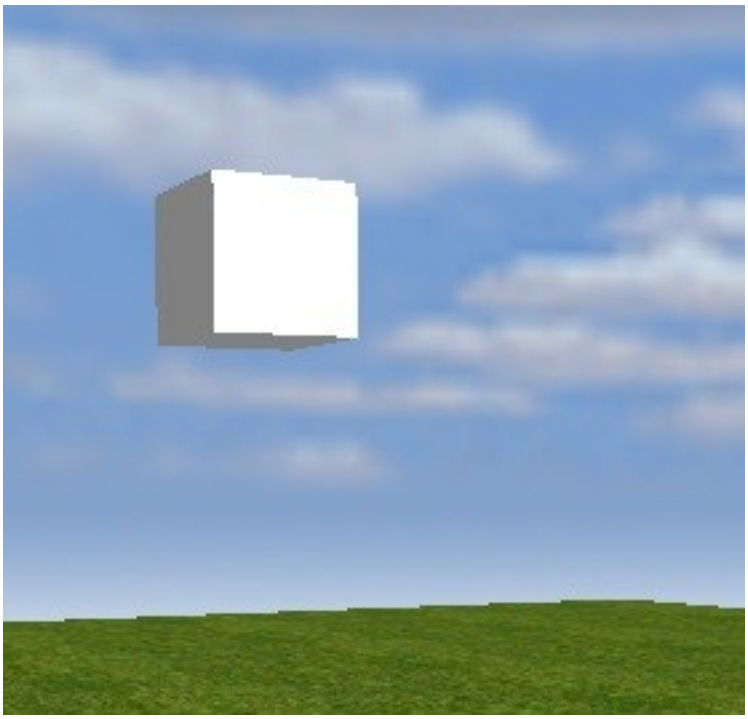
```
TMyCube>>render: ogl
| dx dy dz |

dx := 1.0.
dy := 1.0.
dz := 1.0.
ogl glBegin( GLQuads).
ogl glVertex3f( dx negated, dy, dz).
ogl glVertex3f( dx negated, dy negated, dz).
ogl glVertex3f( dx, dy negated, dz).
ogl glVertex3f( dx, dy, dz).

ogl glVertex3f( dx, dy, dz negated).
ogl glVertex3f( dx, dy negated, dz negated).
ogl glVertex3f( dx negated, dy negated, dz negated).
ogl glVertex3f( dx negated, dy, dz negated).

ogl glVertex3f( dx, dy, dz ).
ogl glVertex3f( dx, dy negated, dz).
ogl glVertex3f( dx, dy negated, dz negated).
ogl glVertex3f( dx, dy, dz negated).

ogl glVertex3f( dx negated, dy, dz negated).
ogl glVertex3f( dx negated, dy negated, dz negated).
ogl glVertex3f( dx negated, dy negated, dz).
ogl glVertex3f( dx negated, dy, dz).

ogl glVertex3f( dx, dy, dz).
ogl glVertex3f( dx, dy, dz negated).
ogl glVertex3f( dx negated, dy, dz negated).
ogl glVertex3f( dx negated, dy, dz).

ogl glVertex3f( dx negated, dy negated, dz negated).
ogl glVertex3f( dx, dy negated, dz negated).
ogl glVertex3f( dx, dy negated, dz).
ogl glVertex3f( dx negated, dy negated, dz).

ogl glEnd.
```

Of course, there are far more concise ways to write this code. Below, we see the result of rendering this object. Since we have not specified any normals for the surfaces, there is no directional light component.



We can modify the code to add the normal like so:

```
TMyCube>>render: ogl
 | dx dy dz |

dx := 1.0.
dy := 1.0.
dz := 1.0.
ogl glBegin( GLQuads ).
ogl glNormal3f( 0.0, 0.0, 1.0).
ogl glVertex3f( dx negated, dy, dz ).
ogl glVertex3f( dx negated, dy negated, dz ).
ogl glVertex3f( dx, dy negated, dz ).
ogl glVertex3f( dx, dy, dz ).
....
ogl glEnd.
```



This would look even nicer if we added a texture to the cube. This requires a bit more setup, as we have to load the texture, and we certainly don't want to do this every time we render the image. So let's add an instance variable to the object called "txtr" and load a bitmap file into it.

```
TFrame subclass: #TMyCube
instanceVariableNames: 'txtr '
classVariableNames: ''
poolDictionaries: ''
category: 'Croquet-Practice
```

The initialize method for our TMyCube class is:

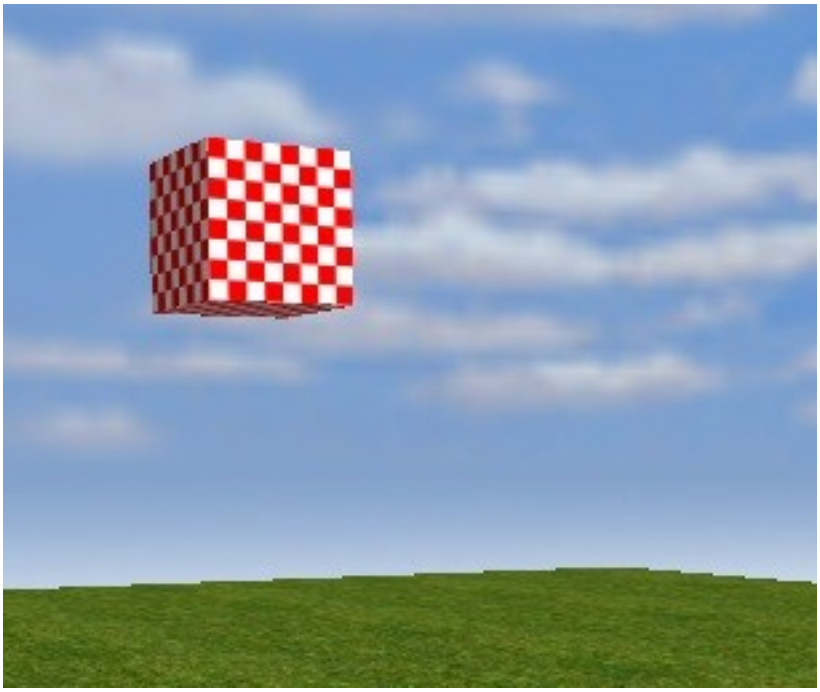
```
TMyCube>>initialize
super initialize.

txtr := TTexture new initializeWithFileName: 'rchecker.bmp'
  mipmap: true
  shrinkFit: false.
```

Now we also have to update the render method to include both the texture as part of the rendering process, but determine how it will be stretched across the cube. The new render method becomes:

```
TMyCube>>render: ogl
```

```
| dx dy dz |

dx := 1.0.
dy := 1.0.
dz := 1.0.

txtr ifNotNil:[txtr enable: ogl].
ogl glBegin( GLQuads ).
ogl glNormal3f( 0.0, 0.0, 1.0).
ogl glTexCoord2f(0.0, 0.0). "1"
ogl glVertex3f( dx negated, dy, dz ).
ogl glTexCoord2f( 0.0, 1.0). "2"
ogl glVertex3f( dx negated, dy negated, dz ).
ogl glTexCoord2f( 1.0, 1.0). "3"
ogl glVertex3f( dx, dy negated, dz ).
ogl glTexCoord2f(1.0, 0.0). "4"
ogl glVertex3f( dx, dy, dz ).
... ..
ogl glEnd.
txtr ifNotNil:[txtr disable: ogl.].
```

And yields the following cube:



Getting Started: Events

The next thing we will do is to enable the cube to respond to a user event. This requires a bit more work, because we have to give the system some hints as to where the cube is when the user picks it with his pointer. Thus, the first thing we need to do is add a simple bounding sphere to TMyCube.

This sphere is used for a number of things. First, it helps the renderer determine if the object is visible or not. If we don't provide a bound sphere, the renderer has to assume the object may be visible and go through all the work of trying to render it, even if it is somewhere behind the camera. By placing the object inside of a sphere that is a very rough approximation of the original object, we know that if no part of the sphere is visible, then there is no way for the object to be.

Second, the sphere is a first approximation test to determine if the pointer, or any other ray, might intersect with the enclosed object. Again, if the ray does not intersect with the sphere, there is no way that it can intersect with the object enclosed inside. Since our cube is a constant size, we can initialize a new boundSphere instance variable at start up.

```
TFrame subclass: #TMyCube
instanceVariableNames: 'txtr boundSphere'
classVariableNames: ''
poolDictionaries: ''
category: 'Croquet-Practice
```

We now initialize the object this way:

```
TMyCube>>initialize
super initialize.

txtr := TTexture new initializeWithFileName: 'rchecker.bmp'
                mipmap: true
                shrinkFit: false.

boundSphere := TBoundSphere localPosition: (0@0@0)
                            radius: (3 sqrt).
boundSphere frame: self.
```

We add a reference to the current object to the boundSphere, which allows us to just add the boundSphere itself to the tests of visibility or intersection. If these are positive results, the object can be easily accessed from the boundSphere for further tests.

We need to add a method that allows external access to the boundSphere as well:

```
TMyCube>>boundSphere
^ boundSphere.
```

The next thing TMyCube must be able to do is determine if the pointer is intersecting with the cube. Keyboard and mouse events are usually vectored to the object that the pointer is over, so we need to know when this happens. The #pick: method is used by the object to determine if the ray is intersecting with it. The #pick: method is not called unless the boundSphere test returns a positive result, so we can actually forgo more precise calculations of intersection and just have #pick: always return true.

```
TMyCube>>pick: aTRay
^ true.
```

The downside of this is that we will get some false positives, because we are now actually picking the sphere that surrounds the cube, and not the cube itself. More complex and precise picking models are available for study in the Croquet source code. See the 'picking' method category of TRay.

Let's have the cube move when we click on it with the pointer. To do this, we first need to let the system know that we are particularly interested in pointer events, and then we need to write a #pointerDown: method. The #eventMask: message is used to let the system know that this object cares about the user clicking the pointer on the object.

```
TMyCube>>eventMask
^ EventPointerDown.
```

Let's have the #pointerDown: move the cube up a bit every time we click on it.

```
TMyCube>>pointerDown: aCroquetEvent
self translation: (self translation + (0@0.1@0)).
```

Getting Started: Simulations

Finally, we get to the point where our cube will start doing something more interesting than just sitting there, waiting for us to do something. We will now have it perform some tricks for us. The simplest simulation we can perform is to base the state of the object on the current island time. Let's add another instance variable called 'move'. We will initialize with the Boolean value of "false".

```
TFrame subclass: #TMyCube
instanceVariableNames: 'txtr boundSphere move'
classVariableNames: ''
poolDictionaries: ''
category: 'Croquet-Practice
```

```
TMyCube>>initialize
super initialize.

txtr := TTexture new initializeWithFileName: 'rchecker.bmp'
mipmap: true
shrinkFit: false.

boundSphere := TBoundSphere localPosition: (0@0@0)
radius: (3 sqrt).
boundSphere frame: self.

move := false.
```

Next, let's modify the #pointerDown: method to flip the state of move.

```
TMyCube>>pointerDown: pointer
move := move not.
self update.
```

Finally, let's add an update method that moves the cube around in a complex circle based upon the island time.

```
TMyCube>>update
| t delta |
move ifFalse: [^self].
```

```
t := self island time.
delta := 3 * (t sin @ t cos @ (t sin * t cos)).
self translation: delta.
(self future: 50) update.
```

This is clearly a simple kind of simulation, where the only information required to determine the render state is the current time. More complex simulations are often based upon previously calculated states and would require significant overhead to recreate this information on every update. We can store previous state in the object, and rely on the repeated #future: messages to keep us from falling too far behind.

Connecting

To join a Croquet session (Island) a participant determines "somehow" the ip/port pair for the router managing the session (island). It then authenticates itself (login), request messages (join), replicated state(sync) and is ready to go.

LAN and WAN differ ONLY by the way in which the discovery of ip/port pair are handled. In the LAN case, each participant aggressively broadcasts the sessions (islands) it hosts so that "neighbours" can pick those up. (this is sub-optimal as we've found ourselves in testing so we'll probably change that slightly.)

In the WAN case, currently no discovery happens. This means that a participant either needs to have a priori knowledge about the ip/port pair of the router or this information must be communicated out-of-band. (again, this is suboptimal and we will add additional discovery services in the future).

How to do it:

Router setup

There are various ways of setting up a router, but they typically look somewhat along the lines shown below. First, we create a dispatcher (so we can maintain multiple routers) and then we add the routers for the sessions we'd like to host, e.g.:

```
"Create the dispatcher"

dispatcher := TExampleDispatcher new.
dispatcher listenOn: 4321. "fixed port"
dispatcher autoCreate: false.

"Create a router"
router := TSimpleRouter new.
router addUserName: 'guest' password: 'guest'.
dispatcher addRouter: router id: sessionID.

"... create more routers here ..."
```

Participant setup

The participant setup follows the general rules except that we provide the participant with all the information it needs initially. The following is the complete script to create a WAN participant which:

a) determines all necessary information,

b) connects to a remote (WAN) router

c) determines whether the session is already active

  c1) if so, joins it,

  c2) if not creates it

d) sets up the initial viewpoint for rendering

Step a:

```
"-------------------- (step a) ----------------------"

"Figure out user name and password for authentication"
user := 'guest'.
pass := 'guest'.

"Figure out address, port, session ID"
address := NetNameResolver addressForName: 'localhost'.
port := 4321.
id := TObjectID readHexFrom:
'ee3320a5924eaf79b1336d2151b80717'.

"Create the participant"
participant := CroquetParticipant new.
participant setup.
harness := participant harness.
```

Step b:

```
"-------------------- (step b) ----------------------"

"Create the controller, connect"
controller := TSimpleController new.
controller connectTo: address port: port sessionID: id.
```

```
"Login, join, sync"
p := controller login: user password: pass. "login
p wait. "until logged in"
p := controller join.    "receive messages"
p wait. "until joined"
p := controller sync. "request replicated state"
p wait.    "until sync-ed"
```

Step c:

```
"-------------------- (step c) ----------------------"

p result ifNil:[
  "If sync failed, create a new island and make me a server"
  island := controller newIsland.
  island future id: id name: 'Test'.
  sync := island future new: SimpleWorld.
] ifNotNil:[
  "Install sync data and retrieve installed island"
  controller install: p result.
  island := controller island.
].
controller beServer.  "act as server"
controller heartbeat: 20. "set heartbeat rate"
harness addController: controller.
```

Step d:

```
"-------------------- (step d) ----------------------"

"Determine the viewpoint for rendering"
entry := controller island future at: #masterSpace.
pc := entry future postcard.
pc whenResolved:[
  harness addIsland: island postcard: pc value.
  sync := harness  viewPortal future postcardLink: pc value.
  sync whenResolved:[
      harness startRender.
      participant openInWorld.
  ]. "ready to render"
].
```

To connect to multiple islands, we simply repeat steps b) and c) as necessary (assuming the connection data is the same). We do not need to repeat step d) since we only need to describe some initial viewpoint (portals etc. will be resolved dynamically).

Signals

Islands are intentionally closed systems. In order to ensure safe and useful replication between replicated Islands, there must be a guarantee that there are no changes that can occur to one that won't occur to another. Further, since Islands are intended to be replicated, references to external objects make no sense because these objects are not necessarily replicated.

Sometimes, however, it is necessary to know that some action or event has occurred inside of an Island without the Island explicitly calling an external method. The way to do this is the Island is able to signal an event, and the external object is able to specify that it has an interest in that event.

Notifications about events occurring in the replicated Island can be obtained using the following expression:

```
anObject runScript: #message when:{aSender. #event}.
```

It is important to note that using events is the *only* "official" way by which an outside observer can get information from a replicated island. There are some other ways (#send: and #get:) but they should be considered debugging aids at best, evil hacks at worst.

To signal the event from inside of the Island we would use either:

```
aSender signal: #event.
aSender signal: #event with: arg1.
aSender signal: #event with: arg1 with: arg2.
```

etc. Note that contrary to other event frameworks, in Croquet the signaler of an event has no idea whatsoever about whether anyone has responded to, or is even interested in, receiving the event. That's a requirement to ensure that we don't get Heisenbugs, e.g., bugs that are triggered by varying behavior depending on whether the event is being observed or not.

Those two methods together give us the infrastructure we need to allow an external object to respond to a change of state of an internal object. This is particularly useful for having external objects track the locations of internal objects, or monitoring the internal state as it changes to update other displays. An excellent example of this is the peer-to-peer chat room system described below in Appendix 1.

Croquet Class Reference

Kernel Objects

TFarRef

A Croquet specific FarRef. Does not allow immediate messages to pass through automatically. Use only in conjunction with #future or #send, e.g.,

```
        ref := island send avatar.
        ref future translation: 100@0@100.
        trans := ref send:[:obj | obj translation].
"<- this will likely NOT be 100@0@100"
```

TIsland

Current representation for a Croquet island. Currently undergoing changes.

TIslandController

I control an island's execution.

Instance variables:

status	<Symbol>	A symbol indicating the status of the controller:
	#closed	- Unconnected
	#connecting	- Trying to connect to a router
	#authenticating	- Authenticating a user
	#ready	- Connected and ready to use.
	#invalid	- Something went really, really wrong.
log	<Stream>	A log stream for debug messages.
island	<TIsland>	The island I am responsible for.
eventQueue	<SharedQueue>	The shared queue holding the messages for the event loop.
eventLoop	<Process>	The process handling messages in the queue.
networkQueue	<SharedQueue>	The queue containing messages from the network.
facets	<Dictionary>	The facets associated with the controller.
mutex	<TMutex>	The mutex used to gain exclusive access to an island.
senderID	<TObjectID>	This controller's sender ID.

TLocalController

An entirely non-replicated, loop-back kind of controller, requiring no network connectivity.

TMessageSend

I am a Croquet message with associated time value, message id, and sequence number.

TMutex

A Mutex is a light-weight MUTual EXclusion object being used when two or more processes need to access a shared resource concurrently. A Mutex grants ownership to a single process and will suspend any other process trying to acquire the mutex while in use. Waiting processes are granted access to the mutex in the order the access was requested.

Instance variables:
Semaphore <Semaphore> The (primitive) semaphore used for synchronization.
owner <Process> The process owning the mutex.

TMutexSet

A TMutexSet acquires all of the mutexes inside it.

TObject

A base class for dealing with some unpleasantness of registries etc.

Instance variables:
myProperties <TObjectProperties> The extra properties for this object.

TObjectID

I represent 128 bit object IDs.

TPromise

I represent a promise being used when a client needs the result of a future message. In that situation the client can use the promise to execute code when the message is actually executed.

Instance variables:

myScripts <Array> Cached script registry for delayed execution
msgId <TObjectID> The id associated with the message.
resolved <Boolean> True if the message has been resolved.
result <Object> The result of the promise (often a FarRef)
resolver <Block> The code to be executed when the promise completes.

Example:

```
island := TIsland new.      "creates a new island"
"creates a promise representing the #new: message"
promise := island future new: Point.
"code being executed when promise completed"
promise whenComplete:[:result |
        result future x: 1.
        result future y: 2.
].
```

Kernel Support

TARC4Cypher

I represent the RC4 stream cypher.

TCryptoRandom

I am a cryptographically strong random number generator. My implementation is based on code found at http://www.cs.berkeley.edu/~daw/rnd/linux-rand.

TFarRefRegistry

A registry for managing TFarRefs

TFileCacheManager

A simple cache manager looking in a particular directory for some cached resource.

Instance variables:
mutex <TMutex> Mutex for concurrent access.
Location <String> Relative uri (using forward slashes) for the cache location.

TLogger

TLogger logs important messages to a file named 'Croquet.log'.

TMessageQueue

I am a message queue sorting elements primarily by their time values and secondarily by their sequence number.

TObjectProperties

I represent some optional state for TObjects which is allocated lazily if needed.

TObjectProxy

I am a serializable name for an object reference.

TSnapshotReader

I am an experimental IslandReader used for replication.

TSnapshotWriter

I am an experimental IslandWriter used for replication.

Kernel Messages

TFutureMaker

I transform messages into future messages.

TMessageData

I represent a single message (datagram).

Instance variables:
receiver <TObjectID> The receiver's id.
selector <Symbol> The message selector.
arguments <ByteArray> The message arguments (encoded!)
sender <TObjectID> The sender's (machine) id.
msgId <TObjectID> The id for this message (to be signaled upon completion)
time <Float> The time (in seconds) when this message should be run
sid <Integer> The sequence id for this message

TMessageEncoder

I encode and decode stuff for TMessageData.

TMessageMaker

I convert messages into future messages.

Kernel Tests

CroquetVMTests

Test suite to verify that the environment that Croquet is running inside of is up to spec.

Router Common

TConnectionDispatcher

This class takes incoming Croquet connection requests and transfers them to the appropriate router. Subclasses must implement the specific mapping required.

Instance variables:

socket	<Socket>	The server socket accepting the incoming connections.
server	<Process>	The process handling incoming connections.
mutex	<TMutex>	Mutex to get exclusive access to the dispatcher.
sessions	<Dictionary>	The mapping from session ID to router.
autoCreate	<Boolean>	If true, non-existing sessions are automatically created.

Class variables:

Default	<TSessionDispatcher>	The default session dispatcher (if any)
Port	<Integer>	The port for the default session dispatcher

TDataGram

I am a simple datagram representation that allows to decouple the facet and the argument data (and reuse the data in many datagrams).

Instance variables:

facet	<TObjectID>	The facet to invoke.
data	<ByteArray>	The arguments for the invocation.

TMessageRelay

I relay messages on a socket.

Instance variables:

address	<ByteArray>	The ip address of the connection.
port	<Integer>	The port for the connection.
socket	<Socket>	The relay socket.
stream	<SocketStream>	The socket stream for easier decoding.
buffer	<ByteArray>	The buffer for send operations.
eventQueue	<SharedQueue>	The queue for scheduling further messages.
outQueue	<SharedQueue>	The queue for outgoing messages.
reader	<Process>	The reader process
writer	<Process>	The writer process
recvCypher	<StreamCypher>	The stream cypher for receiving messages
sendCypher	<StreamCypher>	The stream cypher for sending messages
facets	<Dictionary>	The facets defined for this relay.
recvCount	<Integer>	Number of messages received.
sendCount	<Integer>	Number of messages sent.
recvAmount	<Integer>	Overall number of bytes received.
sendAmount	<Integer>	Overall number of bytes sent.

TMessageRouter

I route incoming messages from croquet clients.

Instance variables:

log	<Stream>	My log stream.
socket	<Socket>	The server socket.
clients	<Array of TMessageClient>	My Croquet clients that I handle.
eventQueue	<SharedQueue>	The event queue for handling messages.
eventLoop	<Process>	The process executing the event queue.
server	<Process>	The server process connecting incoming clients.
facets	<Dictionary>	The facets defined for the router.
lastTick	<Integer>	The millisecond clock value for the last message sent
timeStamp	<Float>	The current time stamp for the island.
autoStop	<Boolean>	If true, close the router when the last client goes away.

TMessageRouterClient

Instance variables:

router	<TMessageRouter>	The router
recvFacet	<TObjectID>	The recv: facet which allows me to receive msgs.

syncFacet <TObjectID> The sync: facet which allows me to act as a server.
serveFacet <TObjectID> The serve: facet which allows me to recv a snapshot.
tickFacet <TObjectID> The tick facet which allows me to get time information.
listFacet <TObjectID> The list: facet that allows me to list other facets.

TMessageRouterTests

Tests for message routers

Router Controller

TRemoteController

This island controller uses a remote message router. For security reasons, a controller is generally set up with two bits of information: a "session key" (used for encryption with the router) and a "list facet" which the controller can use to list other available facets. How to actually transfer those two bits of information is what makes my subclasses and the associated routers special. In the simplest case, we might just use no encryption (no session key) and a well-known list facet. In a more realistic case, however, we would use either encryption or out-of-band techniques (email, https) to get this information across. However, since in general we expect there to be *some* form of authentication, TRemoteController provides the method login:password: as a generic entry point (which then needs to be implemented by the subclasses properly).

Instance variables:
connection <TMessageRelay> The connection to the router
loginPromise <TPromise> Signaled when login completes.
joinPromise <TPromise> Signaled when join completes.
syncPromise <TPromise> Signaled when sync completes.
sentMessages <Dictionary> Measuring the start time of messages.
sentCounter <SmallInteger> A message ID counter.
latencyStats <Bag> A bag full of latency stats.
messageStats <Bag> A bag full of message stats.
cacheManager <TCacheManager> A cache manager to deliver the resources for this island
backDoor <TMessageRouter> The back-door to the message router if hosted on the same machine.

TRemoteControllerConnection

A message relay used as remote controller connection.

Instance variables:
controller <TRemoteController> The remote controller.

TSessionController

TSessionController provides authentication based on the fact that both router and controller share a secret (the password hash) which can be used to initiate a secure connection. To log in, the controller only sends the user name to which the router responds with a challenge - namely an encrypted version of the session key and the list facet (both of which are XORed against the password hash). The controller then requests a list of the available facets which, once the router responds to it, completes the authentication phase. Some interesting notes:
* this scheme never transfers either plain or hashes of passwords over the wire
* the only way to determine whether you responded correctly to the challenge by the router is to see whether the facets are actually listed - if the controller just closes the connection you must assume that password is incorrect

Instance variables:
password <TSecureID> The password hash.

TSimpleController

I am a simplified controller making use of the simplified router.

Router Example

TExampleDispatcher

An example connection dispatcher. Uses a set of existing routers or creates new example routers.

Instance variables:
routers <Dictionary> Maps session IDs to routers.
autoCreate <Boolean> If true, create new routers on demand.
defaultRouterClass <Behavior> The default router class to use.

TExampleRouter

An example router doing no authentication whatsoever and providing everyone with full access. Nice for testing but not much else.

Router Simple

TSimpleRouter

I am a simplified router, which exposes all the facets if provided with the right login facet. I can be used for testing Croquet in an environment where actual authorization schemes haven't been chosen. For now, TSimpleRouter can be provided with a set of user names/passwords or their md5 hashes for authentication.

The model provided here is overly simplified since we NEVER want to give unrestricted access to anything in a real-life use. But for now it is convenient while we're looking for better ways of doing it.

Contacts

TContact

The contact for a Croquet island.

TContactPoint

I am a trivial little contact point for Croquet. I broadcast a single port that another Croquet system can use to connect itself to. I am NOT an industrial strength rendezvous server.

TPostCard

TPostCard is used to find a Croquet router. The method it uses is:

Looks up the routerAddress. If this is a valid address, it will initiate a connection. If it is nil, it will look for the router matching the routerName/ routerID on the LAN.

Once the router has been discovered, the Island is synced.

These instance variables MUST be left as references. Every element in them must exist inside of an Island that may be replicated. There can be no controllers or FarRefs placed into this object. The standin must be constructed by the containing Island.

routerAddress -

routerID - the routerID is a unique ID associated with a single Island with which it should share the same ID and name.
routerName - the routerName is a user created name that can describe the Island/Router group and help the user find the router again.
viewpointName - the name of an entry point into the island. This will typically be a TFrame of some sort.
viewpointID - the ObjectID of the entry point.

Copier

TIslandCopier

I am a facade for copying operations

TIslandCopyData

Represents a blueprint version of data for efficient storage and reconstruction.

TIslandCopyExporter

The exporter is responsible for create a binary blueprint of the object to copy.

TIslandCopyImporter

The importer is responsible for restoring a binary blueprint created by an exporter.

Objects and Croquet Graphics

The Croquet graphics architecture was designed around three principles – simplicity, performance, and extensibility. The design is somewhat different from most graphics engines you might encounter. For one thing, it is much smaller than what you typically see in most game engines. This is because the focus of the design was on the architecture itself and not on the multitude of features that other engines provide. The idea is that if the architecture is simple and extensible enough, it will be easy to add the additional features as needed.

Another difference is that the programmer has direct access to OpenGL throughout the architecture. But this is not just a simple layer on top of OpenGL. It was designed to allow an infinite degree of extensibility at the component/object level, while maintaining a flexible and high performance hierarchical framework within which to place these objects. We sometimes call this a semi-retained architecture, with the idea that it offers all the

advantages of both a retained engine, with its rich structures and internal relationships, while offering the advantages of an immediate engine where any rendering effect can easily be reproduced.

One thing that people are often surprised to learn is that the entire engine (outside of the OpenGL library of course) is written in Squeak! The reason we could do this, instead of resorting to a lower level – higher performance language like C++ is because Squeak is pretty fast already, and it is rarely the bottleneck in good graphics design. More important are the abilities of the underlying hardware and the quality and performance of the algorithms. A well-designed algorithm will almost always outperform a brute force calculation, often by orders of magnitude. Though it is true that a fast algorithm can be made even faster by resorting to low-level coding, this is typically unnecessary, and even when it is, Squeak has some very nice extensions that allow the programmer to turn any piece of the system into a high-performance plug-in.

This section is focused on rendering, but it is important to understand that that is just one aspect of the functionality of a component. Rendering tends to be a good place to start the description, because it has a nice context where the other two component elements, events and behaviors/simulations can be better understood.

The next section is a detailed overview of the Croquet Object graphics architecture.

Object - Fundamental Classes

Here is a list of the major rendering classes and their relationships. This is not a complete list of all Croquet classes, but focuses on the important ones used in a collaborative 3D scene. Indents imply a subclass relationship to the previous class:

TObject *(base collaboration class)*
- TFrame *(base rendering class)*
  - TAvatarUser *(The users interface to the replicated Avatar)*
  - TGroup *(base rendering container class-does not render itself)*
    - TAvatarReplica *(user representative class in a space)*
    - TBillboard *(always faces the camera)*
    - TButton *(3D button object)*
    - TCamera *(scene view class)*
    - TLaser *(visible selection device for Avatar)*
    - TLight *(illumination class - makes everything brighter)*
    - TScrollBox3D *(allows user to manipulate contents of box)*
    - TSkyBox *(displays a rotating sky around a world)*
    - TSpace *(container class, the root of the tree)*
  - TMaterial *(contains the surface properties of an object, can also render)*
  - TMesh *(generic triangle mesh container)*
  - TParticle *(renders particle simple particle system)*
    - TParticleTxtr *(renders textured particles)*
  - TPortal3D *(3D portal class - contains a renderable mini-space)*
  - TPrimitive *(manages render caching of primitive objects for performance)*
    - TCube *(six sided right polyhedron)*
    - TCylinder *(right conic section defined by two parallel circles)*
    - TQuad *(four sided polygon)*
    - TRectangle *(four sided right polygon)*
      - TPortal *(visible link to contents of external space)*
      - TTexture *(bitmap stored for mapping onto surfaces)*
    - TSierpinski *(3D recursively defined pyramid)*
    - TSphere *(a polyhedral sphere defined by a point and radius)*
    - TTeapot *(the canonical Utah Teapot)*
    - TTriangle *(three sided polygon)*
  - TQuadTree *(aids in rendering by using a loose quadtree structure)*
  - TRay *(used for determining distance, orientation, and object selected)*
    - TPointer *(adds event processing to TRay)*

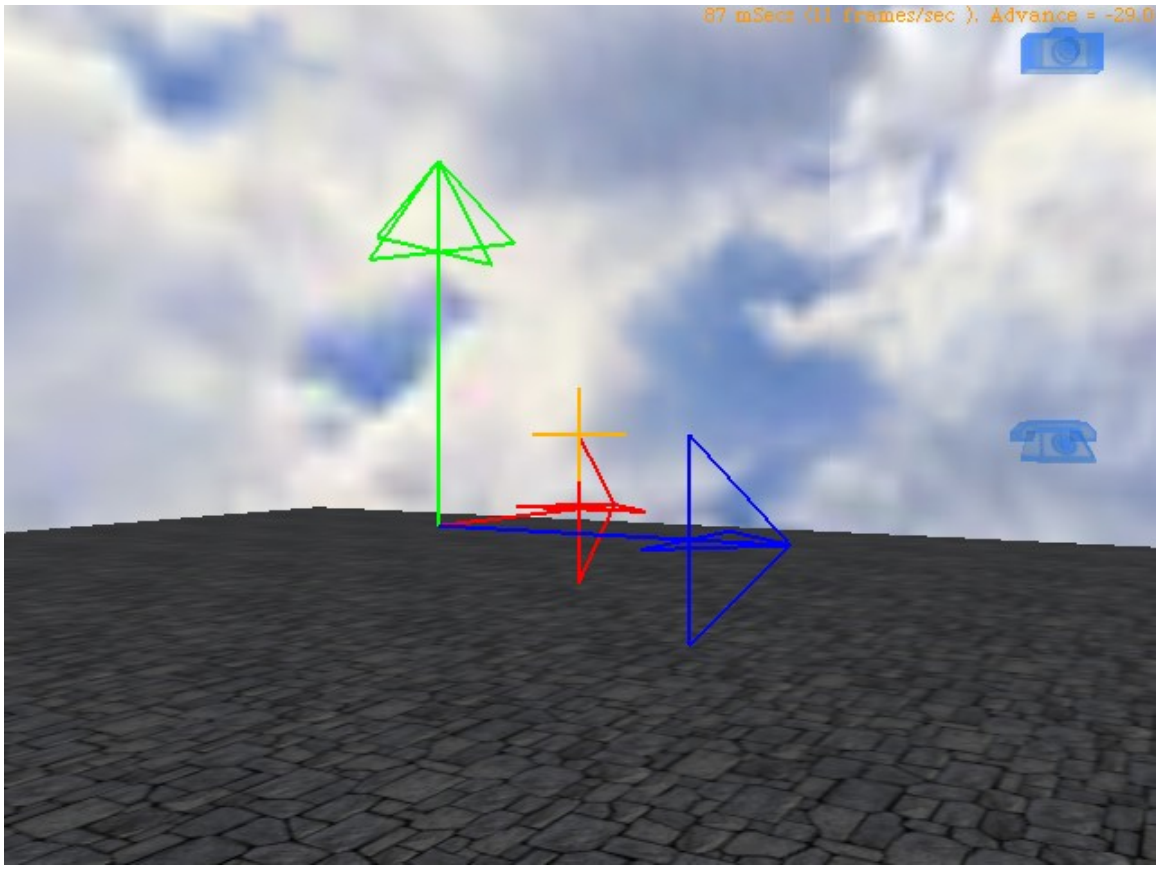
TFrame

Croquet uses a hierarchical structure, or a tree of frames, to manage the rendering of a scene. All of the objects that make up a scene and are embedded in this tree are either TFrames or subclasses of TFrame. TFrame is in turn a subclass of TObject, which incorporates the basic functionality of collaborative objects. This means that the entire rendering architecture, including of course its event processing and simulation abilities, is also collaborative.

All objects exist in a logical tree that has a single root frame, which is a TSpace object. TSpace is a subclass of TFrame, and has additional data managing responsibilities, such as keeping track of the lights in the scene, and rendering all of the transparent objects in the scene after the solid render pass. A TSpace or a subclass of TSpace must always be the root frame of the tree for Croquet to function properly.

We often refer to the nodes in this tree as frames, and in Croquet the nodes are always a TFrame or one of its subclasses. A frame can have any number of children, but it should only have one parent (currently, a frame can actually have multiple parents, but this may be removed at a later date, so don't depend upon this). The parent is the frame that is closer to the root in the hierarchy, while the child is the frame that is further from the root. A good example of this kind of parent/child tree is your hand. If you think of your wrist as the root frame of the hand, then it in turn has one child, which is the metacarpus (that area between the wrist and the fingers has to have a name, doesn't it?) This in turn has five children, which are the bottoms of your fingers. Then each finger has another child, which itself has the final child that is the tip of your finger. The tips of a tree are often called the leaves, though we don't refer to that very often.

From a rendering perspective, a TFrame is somewhat empty. It can certainly render itself, but this rendering is just the arrows pointing along its 3 perpendicular axes.



All of the objects in Croquet have the responsibility of rendering themselves. Then if they have any children associated with them, they need to send messages to each of these, to render themselves in turn. And if they have any children…

This recursive rendering is due to the interactions of three methods: TFrame>>#renderFrame: , TFrame>>#render:, and TFrame>>renderAlpha:. The #renderFrame: method is called first, and it in turn calls the same objects #render: method. If the object has the #hasAlpha flag set to true, it also notes the current global position of the object and saves it and a reference to the current object into a list for use later. It then performs a recursive send of the #renderFrame: message to all of the children of the TFrame object. Once the entire tree is traversed, the root TSpace object traverses the saved alpha list sorted by distance furthest to closest and renders all of the transparent objects in the scene.

What is obvious from this is that the typical way to extend a TFrame subclass is to override the #render: and #renderAlpha: methods. Though you can replace #renderFrame: it is almost never necessary to do this, but when you do, it requires that you understand clearly what you intend to accomplish by doing this, because the system is very reliant on it working properly.

Though TFrame is responsible for rendering itself, it also takes care of positioning itself properly in the space. To do this, it keeps track of two related values, the position or translation of the object, and its orientation in space. A three-element vector defines the translation of the object. A three by three matrix specifies the orientation of the object. We combine these into a single four by four matrix. A good place for an introduction to the 3D transforms used in Croquet can be found in the *OpenGL Programming Guide* by Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, Third Edition. Addison-Wesley. 1999.

TGroup

TGroup is simply a TFrame that is invisible by default. It is usually used as a simple container of other frames. As an example, the 3D window class is a subclass of TGroup with a number of additional frames added, like the window frame (a different kind of frame), the buttons overhead, and the contents are all children of the TWindow class, which has no visible representation of its own.

TAvatarUser

I am the user side of the TUser/TAvatar pair. I handle all of the direct control messages generated by the user and interpret them and forward them to the actual TAvatar. I also check for portal collisions so that I can manage the TAvatar transition from space to space. Finally, I manage how the camera is positioned relative to the TAvatar.

TGroup>>TAvatarReplica

The TAvatarReplica is a frame that represents the users in the scene. It is really not much more than a simple container that the camera tracks as the user moves through the scene, though it does manage collisions and finding the floor.

TGroup>>TBillboard

An object that constantly faces the camera when rendering.

TGroup>>TButton

TButton is an active frame that tracks user events to initiate actions. When the user's pointer is over a child frame of a TButton it is highlighted. When it is clicked, a message can be sent to a target. TButton can have two different child frames to show a boolean state and generate two different messages as it toggles. The TButton itself has no representation, but expects to contain some kind of visible TFrame object that is selectable. This frame (or frames) is

added using the #initializeWithFrame: or #initializeWithFrame:frame2: methods.

TGroup>>TCamera

The TCamera is used to generate an image of the scene from a particular location in a TSpace. It does the majority of setting up OpenGL to render.

The z value of the camera matrix actually points away from the direction of view. This seems to be a standard approach for OpenGL.

instance variables:

bounds
bounds is the area that is rendered into. You can set this (#bounds:) to have an explicit hard value. This is a 2D bounds.

viewAngle
This is the view angle of the camera measured top to bottom. The default is 45 degrees, which means that the angle of the top and bottom edges of the rendered image is 45 degrees.

* zNear, zFar
The near and far clipping planes in the graphics engine.

zScreen
This is the z distance from the camera that the virtual screen would be. That is, when you have an x,y location on the real screen, you can add the zScreen value (x,y,zScreen) and normalize which gives you an accurate pointing vector in 3D space.

length (deprecated)
Computed by initClipPlanes and used to render the test camera.

clipPlanes, clipPlanesTransform
clipPlanes are the four frustum clip planes defined in the cameras local orientation.
clipPlanesTransform are the four clipPlanes in the cameras global orientation frame. This is what is used to determine if objects are visible inside of the view frustum.

viewClip (deprecated)
This is used to test the visibility culling of the camera. It makes the clipping planes visible. This is obsolete, but it may be useful again if we ever start playing with the object clipping code again.

portalClip
This is an additional clip plane defined by a portal. Objects on the near side of the portal need to be clipped away as well. This is a 4x4 transform.

portalPlane
This is a 3D vector calculated from the portalClip.

inPortal
A flag indicating whether we are rendering a portals contents from this camera or not. This is used to suppress certain tests (such as finding floors), and to determine that the avatar should or should not be rendered.

currentSpace
The space we are currently inside of and rendering from.

texture (deprecated)
Simply a texture that gets added to the default rendered camera. Not used anymore unless you really need to see the default camera.

killFrame
A flag to indicate that the current rendered image is junk and should not be rendered. This suppresses a swapBuffers call.

cornerVector
This is a normalized vector pointing to the top right corner of the screen in 3-space. That is, width/2,height/2,zScreen. The cornerVector value is used as an easy way to find a corner of the screen in 3-space. It is computed from the bounds width, height, and zScreen values as follows:

cornerVector := ((bounds width/2.0)@(bounds height/2.0)@(zScreen negated)) normalized.

TGroup>>TLaser

TLaser is used by the TCamera and TAvatar to show other users what this user is currently pointing at or manipulating. This is just a visible representation of the TPointer and does nothing on it's own.

TGroup>>TLight

TLight is used to define the light sources of the 3D world. There can be any number of lights, but only the eight lights that are closest to the user have any direct effect on the lighting at any given time. The reason it is a group is because the light source is usually invisible, and because it can have a visible representation that is made up of a child frames attached to the light.

TGroup>>TScrollBox3D

This is used to manipulate a 3D portal, as well as acting as a base class for the 3D TEditBox.

TGroup>>TSkyBox

TSkyBox is used to render a slowly rotating sky in the world. It is actually a collection of five fullbright textures on a cube (except for the bottom).

dalpha - rotational distance every step.

TGroup>>TSpace

The TSpace is the root class to any scene. It can be subclassed to render something but it is virtually never used this way. In addition to being the root frame, it has a number of responsibilities including managing and initiating the rendering through the portals, managing, enabling, and disabling the TLights that are in the space, initiating the rendering of the tree, obviously by sending its own #renderFrame: message to itself, and finally, it manages and traverses the list of alpha objects that were found in the scene.

TMaterial

TMaterial is used to contain and setup the surface properties of a visible 3D object. It is usually not rendered itself, but it is a TFrame because it can be rendered if the programmer needs to see a material applied independently of the destination surface. When it is rendered, it looks like a teapot with the material on it. The programmer can specify the following properties. For more information on these, consult the *OpenGL Programming Guide*:

ambientColor
diffuseColor
specularColor
emissiveColor
shininess
transparency
cullFace – specify if back faces should be removed
flipFace – flips back to front rendering of faces, usually used for mirrors
fullBright – ignores the above coloring and just renders to the max level
texture – the texture (if any) that will be applied as part of this material

TMesh

The TMesh is used to render complex predefined mesh surfaces. These surfaces are defined by lists of vertices, faces, and associated materials. Typically, a

TMesh will be created from an external file, such as that created by 3D Studio Max. There are additional helper classes such as TLoad3DSMax that are used to construct the TMesh from the 3DS Max .ase files.

Almost all simple arrays are 0 based when indexed by other arrays. Not the materialList.

materialList - this is an OrderedCollection of materials that are used in the mesh. This is what the first element of the faceGroups refers to. This must always be a list.
alias - you can ignore this for now. It is important when we start doing mesh based transforms, because it tells you where the aliased vertices are.
vertices - 0 index based list of 3D vertices.
vtxNormals - the normal 3D vector for the given vertex.
alpha - checks the materials referenced by the facegroups for alpha values. Calculated for you based upon the materialList.
opaque - same as alpha, but inverted.
textureUV - the 2D u,v coordinates of the texture at the associated vertex.
faceGroups - an array of materialList index (indexing starting at 1), and vertex face index arrays (index starting at 0). These are simply interleaved.
boundsChanged - calculated for you at construction time.
boundSphere - calculated for you.
boundsDepth - part of construction and used to determine the depth of the hierarchy.
boundMaterial - only used if you need to see the bound spheres or bound cubes.

These are the analogs to the equivalent TPrimitives. There are two sets because a TMesh may have some materials that have alpha components and some that don't. These are rendered at different times, so they need separate display lists.

materialList - the list of materials used by the different face sets
vertices - the vertices of the mesh
alias - aliased vertex sets
vtxNormals - normals at each vertex
textureUV - the texture uv location at each vertex
faceGroups - collection of face groups (a face group is an ordered list of indices into the vertex arrays)
alpha - flag indicates if this mesh has any alpha values
opaque - flag indicates if mesh has no alpha values
boundsChanged - indicates the boundaries of the mesh have changed
boundSphere - the bounding sphere of the mesh, typically a hierarchy
boundsDepth - depth of the bound sphere hierarchy
boundMaterial - material to render the bound spheres with
cachingEnabled - currently not used, but specifies the OpenGL caching
cachingAlphaEnabled - ""

TParticle

This is a very simple particle system.

size - the length or the particle array, determines max number of particles.
pPosition - B3DVector3Array of particle positions
pVelocity - B3DVector3Array of particle velocities
pAcceleration - B3DVector3Array of particle accelerations
pLifetime - number of seconds (floating point) of life that each particle has left
positionRange - the TBox within which new particles get created (see TBox >> #atRandom:)
velocityRange - the 3D velocity range within which the new particles start life.
accelerationRange - the 3D acceleration range applied to each particle on creation.
lifetimeRange - the lifespan range of each particle.
lastTime - for #the stepAt: method, used to determine time between this and the last cycle.
material - what color are the particles.

To use the particle system you do the following:

```
ps := TParticle initialize: ogl size: 1000
ps setPositionRangeMin:(-0.1@-0.1@-0.1) max: (0.1@0.1@0.1).
ps setVelocityRangeMin:(-1.2@6.4@-1.2) max:(1.2@9.6@1.2).
ps setAccelerationRangeMin:(0@-10@0) max:(0@-8@0).
ps setLifetimeRange: (1500 to: 2000).
```

These values are called inside of the initialize method, but this is how you would set your one.

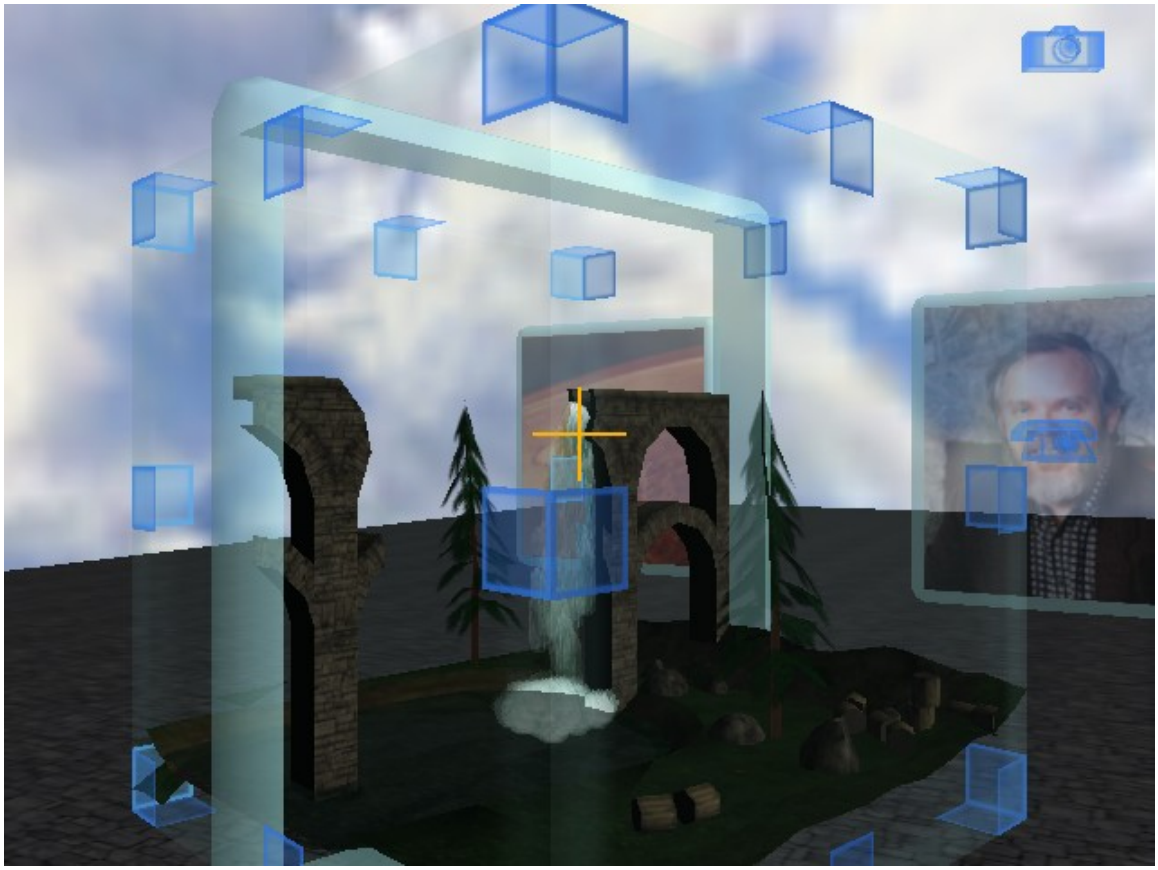
If you want the particle system to move, call self startStepping.

TParticle>>TParticleTxtr

TParticleTxtr is a textured particle

TPortal3D

TPortal3D are 3D portals. These are complete miniature copies of existing spaces that are completely accessible and editable by the user. They are inside of a clipping box, which can in turn be inside of another 3D edit box. 3D portals are extremely useful for getting a complete overview of an environment, even when the user is inside that same environment.



The image above is an example of a 3D portal inside of an edit box. The edit box has the same properties as described above, except that instead of translating the entire contents of the box, the corner rectangles are used to scroll through the 3D space.

TPrimitive

When first starting to learn to develop for Croquet, it is often convenient to simply drop some simple predefined objects into the scene just to see how it is done. The TPrimitive class is the basis of most of these predefined objects. Its major task is actually to manage the internal rendering cache used to speed up the rendering of these objects.

TPrimitive>>TCube

The TCube is actually a regular right polyhedron. That is, each of its corners is a right angle. Other than that, each of its sides can be any length, which means that it doesn't actually have to be a cube.

TPrimitive>>TCube>>TDragCube

TDragCube is just a simple cube that the user can drag around on the screen parallel to the selected side.

TPrimitive>>TCylinder

The TCylinder is a right conic section defined by two parallel circles whose centers are on the same perpendicular line. The TCylinder can be used to make simple cylinders, where the circles have the same radius, or a true cone, where on of the circles has a radius of 0.

TPrimitive>>TQuad

A TQuad is simply a four-sided polygon. There is no requirement for regularity or that the points defining it be co-planar.

TPrimitive>>TRectangle

The TRectangle is a four-sided regular right polygon. It is the base class of both the Tportals and the TTexture classes.

TPrimitive>>TRectangle>>TPortal

TPortal is used to connect and see between TSpace objects. A TPortal has a link to another TPortal (actually any TFrame will do) that in turn is embedded in another space. If the TPortal is linked back to itself, which is the default, it is treated as a mirror.

TPrimitive>>TRectangle>>TTexture

TTexture is a bitmap, or picture that is used in two ways. First, it is used to add a texture to the surface of an object. This can be done directly, or by adding the texture to a TMaterial. TTextures can also be used as 2D pictures dropped into a scene. The pictures can be static, or you can subclass TTexture to create movies, or interactive 2D surfaces.

TPrimitive>>TSierpinski

TSierpinski generates a Sierpinski pyramid.

TPrimitive>>TSphere

A TSphere is a polyhedral sphere defined by a point and radius.

TPrimitive>>TTeapot

TTeapot is the canonical Utah Teapot. It is usually used as a test object for determining the effects of materials, or as simply a placeholder in a scene.

TPrimitive>>TTriangle

TTriangle is a simple three-sided polygon.

TQuadTree

This is a spatial "loose" quadtree. It is used for fast visibility and collision detection.

A loose octree/quadtree is one where an object is contained in only one octree cell. This cell is smallest cell that can completely contain the object, and the one that contains the "center" of the object. In our case, the one that contains the center of the bound sphere. The object is allowed to overlap onto adjacent cells, thus when we test for collisions, we need to compare to the current cell and all of its adjacents. The advantage is significantly simpler and faster bookkeeping.

To use the TQuadTree just use the initialize method:

```
#initializeWithSpace: space frame: frame.
```

This will figure out the proper size of the quadtree from the elements inside of frame and it will place the TBoundSpheres into their proper slots. Then just add the TQuadTree to another frame, usually the TSpace, and you are done.

TRay tests only work if the TRay is a downRay.

The TQuadTree is a hierarchical structure made up of smaller TQuadTrees. If a TQuadTree is empty, its value is set to nil in its containing TQuadTree.
quadCenter - the center of this quad.
quadSize - the 2D x,z size of this quad
quadCorner - the location of the corner
inBox - box containing the centers of objects in this quad
outbox -box containing the entire objects in this quad
radius - radius of the quad
center - center of the quad
spheres - collection of all the bound spheres in this quad
depth - maximum depth of quad tree
qtTL - quad tree Top Left
qtTR - quad tree Top Right

qtBL - quad tree Bottom Left
qtBR - quad tree Bottom Right
boundSphere - bound sphere of the quad
quadOn - flag for rendering. If false, just render normally.

TRay

TRay specifies a position and orientation, usually inherited from its parent TFrame, though you can modify this. It is used to determine the closest object in a particular direction and the distance to that object. Essentially, a TRay is a TFrame with the ability to pick. It uses the local and global orientation matrices to perform the test.

TRay>>TPointer

TPointer is simply a TRay where the instance methods are redefined to convert to TFarRefs or TObjectIDs. It is used as the main user pointer when selecting an object in a scene. TPointer is not intended to be a part of an Island.

TSpace

A TSpace object acts as the root render frame. It is the ultimate container, and is the entry point to a render tree. TSpace objects are contained in CroquetPlace objects, and can be linked via portals.

croquetPlace - the CroquetPlace object that this TSpace belongs to.
color - the default color of the space. This is the color you see when there are no objects to render.
lightFrames - an OrderedCollection of all of the TLights in the hierarchy inside of this space.
portalFrames - an OrderedCollection of all ofthe TPortals in the hierarchy inside ofthis space.
rayFrames - an OrderedCollection of all of the TRays in the hierarchy inside of this space.
alphaObjects - a temporary OrderedCollection of the alpha objects to be drawn in each rendered frame.
currentParent - the current parent object of the frame being rendered. This is used for instanced objects.
currentTransform - the current transform of the frame.
cullBackFaces - this is a flag to turn this on and off for everything in the space.
fogOn fogStart fogEnd fogDensity - fog state variables.
ambientSound - current local sound - to be deprecated.
locator - url object.
dropInFrame - drop the TAvatar at this location when we enter the TSpace.
testRays - turn ray testing on and off (e.g. for portal rendering)

savedAlphaObjects - push the alpha objects list onto this stack if we ever recurse into the space WHILE we are rendering it. This actually happens with the TPortal3D.

Object - Fundamental Support Classes

TBoundSphere

This is the main bounds test object. It is used for object culling against view planes, ray testing, and for collision detection. It is defined hierarchically for the collision detection and object picking algorithms. At the moment, this is one of the slower objects to calculate with complex meshes. We need to turn much of this code into primitives. Also, there are a number of instance variables that may go away or be used in a slightly different way.

Currently, the TBoundSphere global position is set at render time. This is because this is a more efficient method and it also presumes that a TFrame may be instanced in a scene (that is, it may appear as the child of more than one object). If we move to a separate pass for determining ray collisions, then we will no longer be able to instance TFrames, and the global positions of the TBoundSpheres will need to be set by the TFrame when it's own globalPosition is updated.

Other instance variables are:

radius - the radius of the sphere, which should include all of the associated TFrame contents.
radiusSquared - the square radius, here for performance.
children - complex objects often have a need for a hierarchical collection of bound spheres for quick collision tests. These children are also TBoundSpheres.
vertices - if the contained object is a TMesh, then the leaf children contain copies of the vertex information.
box - in computing the bound sphere hierarchy, we utilize TBoxes in an octree subdivision. We keep these around for now, though they can probably be tossed out.
normal - this is to be used by the hierarchical collision detection method. This is the best fit normal to the surface vertices contained in the sphere.
up and side are the two perpendicular vectors to the normal.
offset - to be used by the hierarchical collision method, this is the offset of the normal from the sphere center representing the closest fit to the contained faces.

TBox

TBox has a number of uses. It is an axis aligned bounding box (AABB) defined by two 3D corners. It is used by the TBoundSphere to help define the boundSphere tree. It is used by the TParticle as a parameter bounds for it's instance variables. It is also used by the TQuadTree for it's hierarchical representations.

TForm

TForm is an imaging object that maps Squeak forms into OpenGL textures. It is also intended to act as a static Croquet object type in that it will typically not be stored as part of an Island but must be loaded either from a local cache, a remote cache, or from another user. It can also generate a thumbnail version of itself that can be copied and stored as part of a TTexture.

sha - the Secure Hash Algorithm value from the original file
form - the Squeak Form object.
updateRect - what part of the OGL texture to update.
extent - the actual original extent of the texture.
bMipmap - boolean flag, determines if mipmapping is on.
bShrinkFit - boolean flag, determines if we shrink or grow the texture to fit the proper power of two extent.
extension - extension bit flags for further bitmap processing
memUsed - the memory required to store the OGL texture.
objectID - a TObjectID
bThumb - indicates whether or not this TForm is a thumb. If it is, then it needs to be resolved to the "real" TForm, which may be either cached or located in a remote server.

NOTE: Dynamically generated TForms (those not loaded from elsewhere) are the same as themselves. That is, they cannot really generate thumbnails.

TFormFind

TFormFind is used to find an image file either locally in cache, or remotely in server, or from another user via the router. It is only used by the TFormManager and is created inside of TFormManager>>#resolve:distance:

sha - sha name used for caching files locally.
url - web based access of object
routerAddress, routerID - used to find the router to ask other users for file.
distance - distance from the camera, used to prioritize search.
bMipmap - is the result mipmapped?
bShrinkFit - is the result shrinkfitted?
extension - any extensions, such as color key?

TFormManager

The TFormManager is created in the CroquetHarness, but is also used by the OpenGL object. The reason for this is the harness is used to create Islands and their contents, and OpenGL is used to render the contents, hence the TForm objects used to generate textures must be accessible from both.

TFormManager does a number of things:

- When rendering, it will look up the real TForm in the textureDictionary when provided with a TForm thumbnail.
- If a TForm is requested that is not in the textureDictionary, TFormManager will attempt to load a new copy in the following order:
    > Check the cache directory for a local copy with the name to the sha
    > Check the URL provided as part of the thumbnail object
    > Request a copy of the texture from the originator of the island.

- When it loads a new TForm it does the following:
    > Computes an SHA hash from the file
    > Generates a new TForm object that is placed into a Dictionary that is accessed by OpenGL
    > Loads a bitmap file into a Form that is part of a TForm
    > Generates a TForm thumb object and places the SHA hash into it. This is what is actually returned from the constructor.
    > Saves a copy of the file into the local cache with the name <sha hash value>
    > Saves a copy of the file into the remote super cache with the same name.

TLoad3DSMax

TLoad3DSMax Notes:

There are two parts to the class - the parse routines and everything else. The actual #parse: method creates a block hierarchy - essentially a kind of parse tree, from the inputs. 3DS is somewhat regular in its construction, but there are a few things to be careful about. The tree that gets constructed is made up of a field name and a field. The field names are tokenized 3D Studio field names and the field is the child tree or the actual text field data - not parsed into actual numbers or anything yet. One improvement on performance might be to interleave this step with the next.

Once the tree exists, a second pass is made that converts the text fields and constructs the actual frame hierarchy. The field name of each node is matched and the appropriate #make****: routine is called on its contents.

Once a raw mesh is set up inside the #makeGeometry: method, the #reconstruct: method is called. This does a lot of massaging and optimization of the raw vertices - including aliasing, etc.

Anyway, start at:

#initialize: fileName: scale: shadeAngle: textureMode:

Based upon the field names in the tree constructed in #parse: you just make the call to the next #makeXXX:.

- Though we have all of the transforms for the hierarchy, the actual locations of the meshes are already transformed. If I can think of a good reason to un-transform the mesh elements and then add the transform to the actual frame, I will do it. For now, treat it as a solid body.

- The texture rotations are face centered. This requires an offset of 0.5, the rotation, and then putting it back. Not sure if I really want to pre-transform the texture uv coordinates. Also, not sure if this is already done, as the meshes are.

- This class will act as a template for importers, though they all seem so different that this may be easier said than done.

- This object is actually never rendered, and really doesn't need to be a subclass of TFrame.

TLoadMDL

TLoadMDL is used to load Alice models.

TRenderAlpha

This class is used to render the alpha (transparent) objects. As an alpha object is found, a TRenderAlpha object is created that includes:
tObject - the object
transform - the object's OpenGL global transform
distance - the distance from the TCamera used for sorting
parent - its parent frame (if needed - until we remove multiple parents from the system)

TSelection

TSelection is a holder for data generated by the ray testing in Croquet. See TRay.

target - this is the frame object owner

frame - this is the frame object that is selected.
framePosition - this is the ray location relative to the selected object.
parent - this is the parent of the selectedFrame at the time of rendering/selection.
parentCameraTransform - this is the transform of the camera frame at selection time
distance - this is the distance from the ray position.
distanceSquared - this is the squared distance from the ray position.
index - this may be used by the selectedObject. It is usually the surface index.
normal - this is the surface normal at the selectedPoint.
point - this is the selected point in the local selectedFrame reference.
triangle - used by TSelectionTracker
cameraTransform - the camera global transform at the last time of selection.
scale - the current scale at which we picked this object. Used by TPortal3D.

TXSelection

Same as TSelection except we ensure that we never have access to an object that is inside of an Island. TXSelection (eXternal Selection) either has a copy or a has a reference ID to the original object.

Window

TWindow

TWindow is the Croquet windowing system. It can handle both 2D and 3D contents. Objects that are primarily 2D can be added via #contents:. 3D objects that require the removal of the back face of the window can be added via #contents3D:.

TWindowButtons

TWindowButtons is a frame that creates and positions the buttons at the top of the window.

TWindowFrame

TWindowFrame is the outside control frame for the TWindow class.

Islands

RecurseWorld

RecurseWorld is used to illustrate the construction and population of a new Island. It is designed to be linked with the SimpleWorld Island.

SimpleWorld

SimpleWorld is used to illustrate the construction and population of a new Island. It is designed to be linked with the RecurseWorld Island.

Harness

CroquetEvent

The CroquetEvent includes a complete description of a user event including which mouse and keyboard buttons have been selected, a complete description of the 3D target objects inside of the selection, and the avatarID. It is the object passed to the Croquet objects to act upon when an event occurs.

CroquetHarness

Croquet Harness is the minimal interface to the underlying support infrastructure. It is used to manage changes in screen real estate, and track events to vector to Croquet.

userID - a unique TObjectID which identifies a particular user.
dispatcher -
controllers - collection of controllers that we have created that are connected to both the local and remote routers.
contactPoint - finds all of the local broadcasting routers via their own contectPoints.
formMgr - the default TFormManager. used to manage TForms.
cacheMgr - default TFileCachManager.
avatar - TAvatarUser
ogl - the OpenGL object, which is the interface to OpenGL.
viewPortal - the main viewing portal into the current space of interest
postcard - a TPostcard that is a reference for the rendered viewpoint.
bounds - the 2D bounds of the viewing space
systemOverlayPortal - overlay portal supporting the system overlay space
systemOverlay - system overlay space
systemIsland - the non-replicated Island containing the system overlay content
readyToRender - semaphore flag indicating that we can now render
renderProcess - a render process fork.
doRender - boolean indicating we can begin rendering
viewPoints - a listing of all the FarRe f viewpoints that are referenced by portals.
eventPointer - a TPointer used during rendering to determine user interactions with 3D objects.

event - a CroquetEvent object used to pass events to event processing objects inside of Croquet
yellowButtonPressed - boolean indicating yellow mouse button is pressed.
redButtonPressed - boolean indicating red mouse button is pressed.
overlays - array of overlay spaces passed to system to be rendered.
islandsByName - Dictionary used to look up Croquet Islands by their name.
islandsByID - Dictionary used to look up Croquet Islands by their ID.
enableIslandCache - enables checkpoint caching of Islands to disk.
snapshots - create image snapshots of current space.
windowData - holds a copy of a generic TWindow.

CroquetMaster

A simple Morph used to manage a minimal version of Croquet. CroquetParticipant is actually the superclass for this class and performs the majority of the work in setting up a new Croquet environment.

CroquetParticipant

CroquetMorph is currently the jumping off point for Croquet. We use Morphic to handle low-level events and communicate with Squeak. At some point, this dependency will change.

Appendix I – Peer-to-peer Chat in Tweak

Andreas Raab

Design issues

There are two fairly distinct use cases for implementing peer-to-peer chat inside of Croquet. Since the use cases have very different security implications, let me describe them individually: The first use case is that of an IRC chat model, where a space is a chat room in which people can talk to each other. Since it's a public forum, everyone can listen in and (eventually) talk back. From the security point of view everything is open and accessible and our users are (hopefully) aware that everything they say will be relayed to everyone in the room.

The second use case is that of me talking to another person or a specific subset of people. The main difference is that in terms of security my expectation is that the conversation I have is held privately between that person and myself. The main issue here is that Croquet -as an architecture- ONLY supports the first model (public chat room) with respect to its participants and that the latter (private chat) requires special care (actually up to the extent where one might question whether it is worthwhile to use Croquet directly or simply hook into an out-of-band IM service).

The reason for this property is that replicated computation means that anything you inject into the replicated space *must* be broadcast to every participant. This is not an optional feature, it's a basic requirement for the architecture. Similarly you as the application designer *must* therefore assume that someone will be using their own custom Croquet client which is capable of recording and analyzing all messages sent to the replicated space. It's like a packet sniffer on steroids.

Because of this, the best option to preserve privacy is often to simply not to use an environment that everyone and his brother have access to. For example, given that I have enough information about the person I want to talk to, I could send them an invitation to join a "private chat space" that I host and where I only let those people in. But since that may not be feasible at times an alternative is to use a pseudo-VPN approach where messages are encrypted with the public key your avatar carries along and therefore can only be decrypted by your private key (beware though that an attacker might be able to replace your public key with his so there should actually be some out-of-band verification of the public key).

Implementation issues

In the following, I will make the assumption that we have a way of securing "private" messages that are only intended for a user (or a subset of users) in the space. This may mean public key encryption, or it may mean that we simply assume that the clients are not compromised and filter properly based on the intended audience (this is just to cut out the whole security implementation issues which are a different beast altogether and not part of this discussion).

Given the above, I can describe the whole chat implementation by two operations:
1. Obtain a message from the user and send (replicate) it
2. Get a notification about a new message and relay it to the user

Since the second operation (receiving) really sets up the requirements for the first, let's start there. Notifications about events occurring in the replicated can be obtained using the following expression:

```
anObject runScript: # message when:{aSender. #event}.
```

[BTW, we are now getting into the muddy waters of the unfinished Tweak/Croquet messaging integration. In this memo I will consistently use the things that work *right now* (like #runScript:when:) even though they will be considered horribly broken a month or two from now and it will look horribly complicated too].

It is important to note that using events is the *only* "official" way by which an outside observer can get information from a replicated island. There are some other ways (#send: and #get:) but they should be considered debugging aids at best, evil hacks at worst.

To signal the event we would use either:

```
aSender signal: #event.
aSender signal: #event with: arg1.
aSender signal: #event with: arg1 with: arg2.
```

etc. Note that contrary to other event frameworks, in Croquet the signaler of an event has no idea whatsoever about whether anyone has responded to, or is even interested in, receiving the event. That's a

requirement to ensure that we don't get Heisenbugs, e.g., bugs that are triggered by varying behavior depending on whether the event is being observed or not.

Those two methods together give us the infrastructure we need for a replicated chat. The one remaining issue is that we need to agree on a common "meeting point" for the chat messages to arrive (e.g., the "sender" for the event in the above). For simplicity, I'll just use the space itself in which the avatar is in (this is not a good thing since we probably want more infrastructure for channels etc. but it's simple to point of being trivial for this exercise).

This gives us the basic mechanism to use for our replicated chat. All we need to do is the following:

```
TAvatarUser>>addToNewSpace:aSpace

  "Add this avatar to aSpace"

  "unregister myself from the old space"
self stopScript: #recvChatMessage:for:.

   " .... lots of stuff here .... "

  "register myself with the new space"
  self runScript: #recvChatMessage:for: when:{aSpace.
#chatMessage}.


TAvatarUser>>sendChatMessage: msgString to: recipients
  "Send the given message to a set of recipients.
   For simplicity we'll assume that the msg string is
   already secured (encrypted whatever)"

  "Note that we're being lazy here by signaling the event
   directly. In a real implementation you'd have a chat object
   with an API and that chat object would decide which event
   to use for what. The client shouldn't just wildly signal
   idiosyncratic events since they might conflict with other
   uses that this particular client knows nothing about."

   currentSpace future
      signal: #chatMessage
```

```
    with: msgString
    with: recipients.

TAvatarUser>>recvChatMessage: msgString for: recipients
 "Receive a message for the intended list of recipients"
 (self isRecipientIn: recipients) ifTrue:[
  "This message is for me, so relay it to the user.
   Signal another event so that a client can just observe
   the chat events in the avatar and doesn't have to deal
   with (changing) spaces, filtering etc."
   self signal: #chatMessage with: msgString with: recipients.
 ].
```

Note that at this point we have a fully functional chat API that any client (regardless of whether it is written in Croquet, Tweak, Morphic, or MVC) can use. All you need to do is to get your hands on the harness' avatar, send it a chat message with the list (or channel name) for the intended recipients and register yourself to receive chat messages broadcast to your avatar in return.

Nasty Tweak Details

Finally, a few comments on some nasty Tweak implementation issues. As you have noticed, there are some evil, evil things going on at times with the current lack of integration. Generally, if you need interactions between Tweak and non-tweak computations, you should use events to get input into Tweak (as you noticed, otherwise you might be in an odd intermediate state). You should therefore set up another event response via, say:

```
TweakWorld>>onChatMessage: msgString
 "Open a message chat"
  <on: chatMessage in: avatar>
```

The one thing you *may* need is an evil little hack to fix up the default hand for the computation (you'll rapidly notice if you need this) via (in the very same method):

```
Processor activeProcess hand: Processor activeProject
primaryHand.
```

Acknowledgments

With all honesty, Croquet was built upon the shoulders of giants. In particular, the existence and power of Squeak allowed us to convert our dreams into a reality that exceeded even our expectations. Many thanks to the original Squeak Central team for providing us both an optimal environment within which to create such a beast, and for the help and encouragement they provided throughout the process. In particular, thanks to Dan Ingalls, Ted Kaehler, Scott Wallace, Andreas Raab (also a member of the Croquet team) and John Maloney. It is no exageration when we say, "We couldn't have done it without you."

The Croquet Architects would also like to thank Kim Rose for her enthusiastic support and relentless attention to ensuring that our trains continued to run on time. She may have had the hardest job of all of us.

Chris Cole's belief that the world needed a change and his willingness to back up this belief with his substantial support was critical. It allowed us to take the big jump to focus on creating something new, very different, and certainly not "politically correct".

Thanks to Chunka Mui and DiamondCluster International for their substantial and ongoing support and willingness to share ideas.

Thanks to Bran Ferren and Danny Hillis and the staff at Applied Minds, Inc. for providing contacts, advice, much needed support, and a roof over the heads of those at Viewpoints Research Institute.

Thanks to Michael Moody and Peter Maguire for the great content and worlds. Most of the 3D models you see in Croquet were built by Michael and Peter.

Thanks to Michael Rueger for naming Croquet, setting up the website, and for his ongoing technical help.

Thanks to Patrick Scaglia and HP for their extraordinary support during a critical time of incubation of the project. Croquet is an extremely complex system. Patrick gave us the room to really think things through and build it right.

Thanks to Rick McGeer for his invaluable role as evangelist, cheerleader, and sage. Rick bought in early and was crucial in forming the early support that we now enjoy.

Thanks to Howard Stearns and everyone at the University of Wisconsin and the University of Minnesota and the rest of the Croquet Consortium. These were the people that bought into the idea of Croquet before there was a Croquet.

Thanks to Julian Lombardi and Mark McCahill. It was clear we shared a common path from the beginning. They brought a huge number of resources to the Croquet project, but most importantly, they brought themselves.

Thanks to Greg Nuyens, for taking up the difficult task of leading us to take Croquet to the next level.

Thanks to those that contributed to this document and helped proof it. In particular, Howard Stearns, Greg Nuyens, Kim Rose, Mark McCahill and Andreas Raab contributed invaluably to this effort. I, of course, retain sole ownership of all remaining errors.

And of course, thanks to my main partners in this journey, Andreas Raab and David Reed. Andreas has proven to be a match for any crazy idea I have come up with and has shown himself to be one of the great system designers. His insight, creativity, and ability were essential to us realizing this dream. David and his ideas are at the very core of Croquet. To think that an idea that David had thirty years ago would enable a system like Croquet today is astonishing. But more than just coming up with great ideas, David has been a key system builder, steadfastly drilling where the wood was thickest, and has been incredibly patient as a teacher and mentor as we sought to understand the deep ideas that were so clear to him. It is easy to stand on the shoulders of giants when they happen to be your colleagues.

Building any new architecture is a stressful endeavor, but this one was particularly so. The economic situation constantly threatened to derail us, and the project demanded an incredible degree of imagination coupled with technical rigor. Without the support of our families and friends, their belief in us, and their patience and love, Croquet would have remained an idea for the future.

Finally, we would like to thank Alan Kay. Alan put everything he had into his belief in this project and us. At various times he acted as financier, evangelist, roady, cheerleader, coach, and rainmaker. He was responsible for putting the Croquet team together and making room for us to succeed. Perhaps we can repay him by doing the same for a future generation of dreamers.

Useful References

Online

Croquet Website: http://www.opencroquet.org/

Wikipedia entry: http://en.wikipedia.org/wiki/Croquet_project

Squeak Website: http://www.squeak.org/

Squeakland Website: http://www.squeakland.org/

Viewpoints Research Institute: http://www.viewpointsresearch.org/

OpenAL Website: http://www.openal.org/home/.

OpenGL Website: http://www.opengl.org/

Open Dynamics Engine Website: http://www.ode.org

Hard copy

OpenGL Programming Guide, Third Edition. OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, Addison-Wesley. 1999.

OpenGL Game Programming. Kevin Hawkins, Dave Astle. Prima Publishing. 2001.

Croquet - A Collaboration System Architecture. Smith, David A., Alan Kay, Andreas Raab, David P. Reed. C5: Conference on Creating, Connecting and Collaborating through Computing. 2003. p.2-.

Squeak: Open Personal Computing and Multimedia ed. Mark Guzdial and Kim Rose. Prentice Hall, 2002

Squeak: Object-Oriented Design with Multimedia Applications. Mark Guzdial. Prentice Hall, 2000.

Robot Manipulators: Mathematics, Programming, and Control. Richard P. Paul. Massachusetts Institute of Technology. 1981.

Index