
CROQUET.....



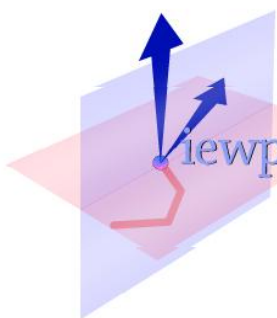
THE USER MANUAL

DRAFT REVISION 0.1

CREATED BY:

David A. Smith
Andreas Raab
David Reed
Alan Kay

OCTOBER 18, 2002



Viewpoints Research Institute

1209 Grand Central Avenue Glendale, CA 91201 tel. (818) 332-3000 fax (818) 244-9761

Copyright © 2002 by Viewpoints Research Institute

Notice

Croquet is a work in progress. Everything in this document is subject to change, including the name of the system, user interfaces, and APIs. You may use and modify the system described herein in any way you like, but be aware that we reserve the right to modify it, and further, there is no obligation on our parts to support or maintain this system or any resulting code that might be developed by you or any other third parties. In other words, use at your own risk.

Note also that start-up times are extremely long. This is because we aren't properly utilizing the dynamic world loading facility yet, and we are loading many example environments into the system at the start. Please be patient.

That said, it is our full intention to make Croquet into as high a quality a product as you will find anywhere, commercial or not. Until a better license is written, Croquet is covered under the current Squeak license.

Team Croquet

Table of Contents

| | |
|---|----|
| Notice | 2 |
| Table of Contents | 3 |
| 1. Introducing Croquet | 4 |
| 2. Getting Started..... | 9 |
| 3. Exploring Croquet | 15 |
| 4. Picking and Manipulating – Croquet Windows | 18 |
| 5. Spaces and Portals | 24 |
| 6. Navigator Dock | 29 |
| 7. Additional Controls | 36 |
| 8. Croquet: Peer-to-peer | 39 |
| 9. Scripting in Croquet | 43 |
| 10. An Introduction to Programming with Croquet | 59 |
| 11. TeaTime: A scalable real-time multi-user architecture | 66 |
| Appendix A: Open Items..... | 69 |
| Appendix B: Is “Software Engineering” an Oxymoron? | 72 |
| Appendix C: Biographies | 79 |
| Index | 83 |

1. Introducing Croquet

Croquet had the working name of Tea until recently. You will see many references to Tea in the system, in the code, and even in this document. Just assume that when you see Tea, we mean 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. 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 in 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 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 they 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 does this.

Croquet is...

Croquet is a computer software architecture built from the ground up with a focus on deep collaboration between teams of users. It is a totally open, totally free, highly portable extension to the Squeak programming system. Croquet is a complete development and delivery platform for doing real collaborative work. There is no distinction between the user environment and the development environment.



Croquet is a multi-use collaboration environment where there are no limits to what the user can do.

Croquet is also a totally ad hoc multi-user network. It mirrors the current incarnation of the World Wide Web in many ways, in that any user has the ability to create and modify a “home world” and create links to any other such world. But in addition, any user, or group of users (assuming appropriate sharing privileges), can visit and work inside any other world on the net. Just as the World Wide Web has links between the web pages, Croquet allows fully dynamic connections between worlds via spatial portals. The key differences are that Croquet is a fully dynamic environment, everything is a collaborative object, and Croquet is fully modifiable at all times.

The current computer user paradigm is based upon a completely closed individually focused system. The user has a very low-bandwidth communication capability via e-mail,

or instant messaging, but outside of some very simplistic document sharing capabilities, the user is quite alone on his desktop.

Croquet has been focused on high bandwidth collaboration from its inception. Simply put, the fundamental building block of the Croquet architecture is a system that makes every single object in the system collaborative.

One way to think of the Croquet environment is as a high bandwidth conference phone call. Once a connection is made, the user not only has voice communication with the other participants, he also has the ability to exchange documents, collaboratively design systems, perform complex simulations, develop complex project plans, and manage complex projects.

Squeak

Squeak is a 21st century dynamic-object wide-spectrum operating and authoring environment derived from the 1970s Xerox PARC Smalltalk system in which overlapping window GUIs, Desk Top Publishing, media authoring, and many other familiar software systems were first developed. Several of the authors of Squeak were principals at Xerox and were co-creators of many of the PARC inventions.

This preliminary document assumes a reasonable familiarity with Squeak and its development environment. There are a number of online and hard copy references to Squeak that are readily available.

Croquet was developed entirely within Squeak, both as a language and as a development environment.

[more to come]

Thanks to...

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 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. No lie when we say, we couldn't have done it without you.

We 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 for their substantial and ongoing support and willingness to share ideas.

Thanks to Bran Ferrin and Danny Hillis and the guys at Applied Minds for providing contacts, advice, much needed support, and a roof over the heads of Viewpoints.

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 Bill Cole for helping to set the agenda for the next big steps, and for his patience.

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

Of course, thanks to everyone that has contributed money, ideas, and time to this and the related projects.

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.

2. Getting Started

Before You Start

Croquet requires a reasonable hardware graphics capability. If your computer is less than two years old, you probably have a good system to run on. Otherwise, you might want to consider acquiring a new graphics card. An nVidia GeForce 2 or better will work quite well and is reasonably inexpensive. Croquet does run on a Macintosh computer, but you may be more limited in graphics card choices.

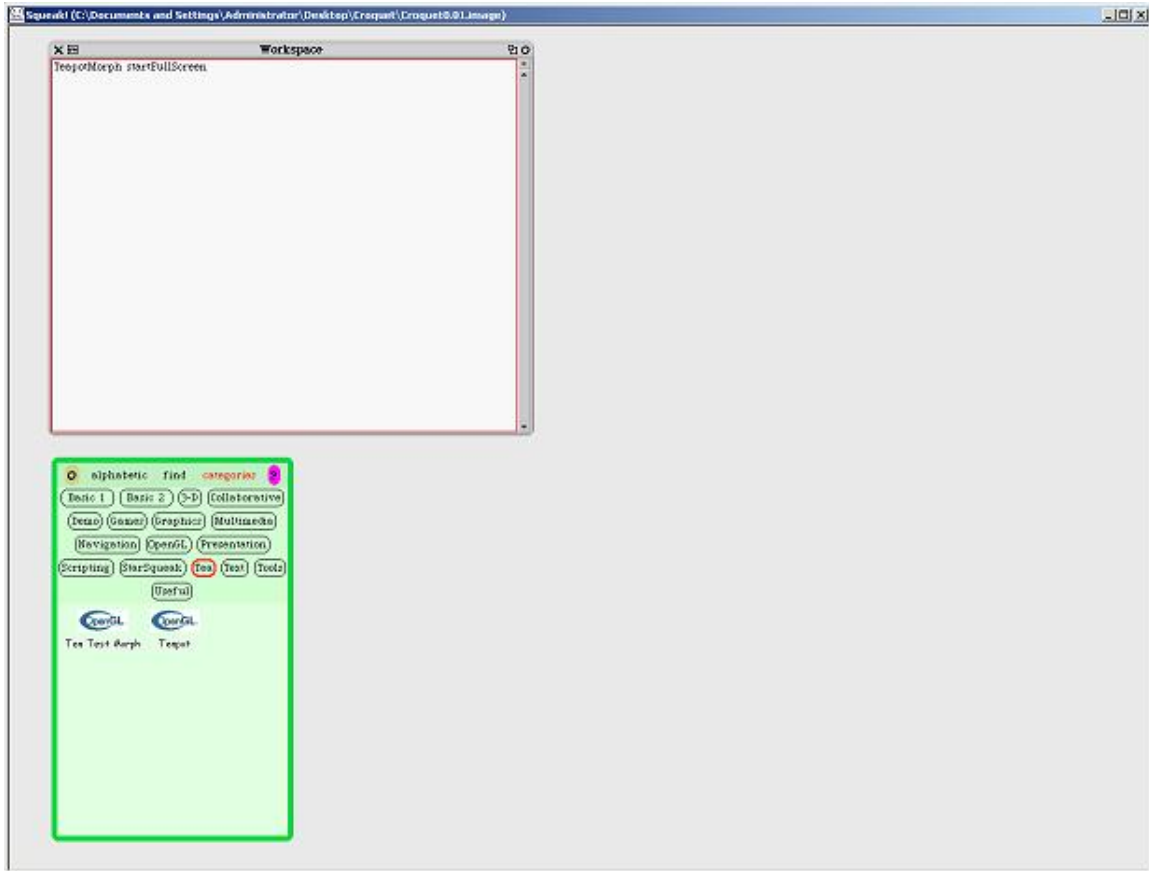
Croquet requires that you set your graphics color depth at 32 bits. We do not currently set this for you, so you will need to modify this yourself. On Windows, you can select the “Display” control panel (or right click on the Windows desktop and select “Properties” from the menu). Select the “Settings” tab, and set the color to “True Color (32 bit)”. Now you are ready to start.

On the Macintosh [what you do on the Mac goes here.].

Starting Croquet

At present, Croquet runs inside of a Squeak project. At some point in the near future, you will be automatically launched directly into the Croquet.

Once you have downloaded the Croquet files, simply double click on the TeaSqueak1.9.exe file (or if it is a Mac????). This will launch you into Squeak, and you will see the following screen:

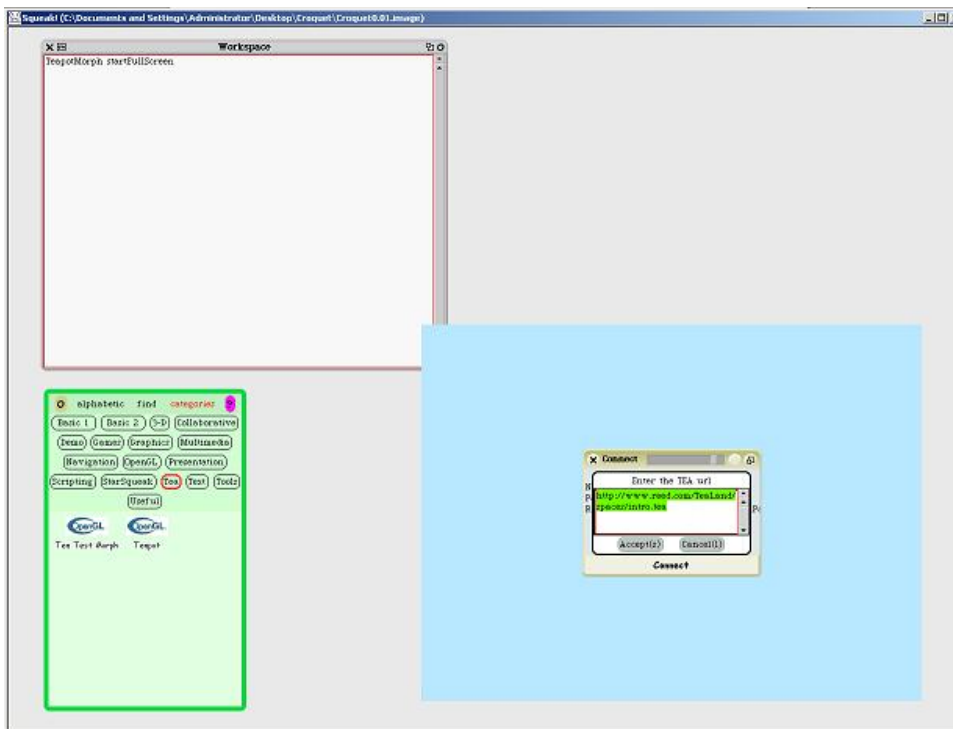


Typical startup screen for running Croquet.

At this point, there are two ways to start up Croquet. The first and easiest is to simply execute the line *TeapotMorph startFullScreen* in the Workspace. To execute a line of text in Squeak simply press Alt-D in Windows. Croquet will start up in full screen mode. The second way to start is to drag and drop the icon labeled “Teapot” in the green objects list.



This is what you will see if you execute “TeapotMorph startFullScreen”.



And this is what you will see if you drag and drop “Teapot” icon to the lower right of the screen.

There is a dialog box on top in the center of the screen “Enter the Tea url”. Press the “Cancel” button for now. CAUTION: Pressing Accept is not completely supported yet.



If this is the first time you have run Croquet, the system will now begin converting the content files into its native object format. This may take some time, please be patient.



Once Croquet has loaded, you will see something like the image here. The window on top of the screen is used to connect to other users. For now, simply close the window at the (x) beside the “Connect” window name.


Exiting Croquet

Before we go any further, a word on exiting from Croquet. Croquet is actually running as a Squeak morphic window. To exit this, you can click the middle mouse button or ALT-click the left mouse button almost anywhere inside of the Croquet world. Anywhere except a project window that is. You will see a halo of small circular buttons around the Croquet window that looks like this:



or perhaps like this:



Simply click the  button and the window will go away. To exit Squeak itself, click anywhere on the desktop to get the following menu.



Just select the last menu item labeled "quit".

3. Exploring Croquet

We are now ready to begin our exploration of Croquet. This description relies on a three-button mouse for navigation. Use equivalent modifier buttons on the Macintosh.

As you can see below, we should find ourselves inside of a real time 3D environment with a number of windows floating in space before our eyes. Croquet uses two primary mouse buttons. The left mouse button is for picking and the right mouse button is for moving around inside the space. One thing to be aware of is that a programmer can create any kind of user interface he wants, so a game world may have a very different way of moving through the world.

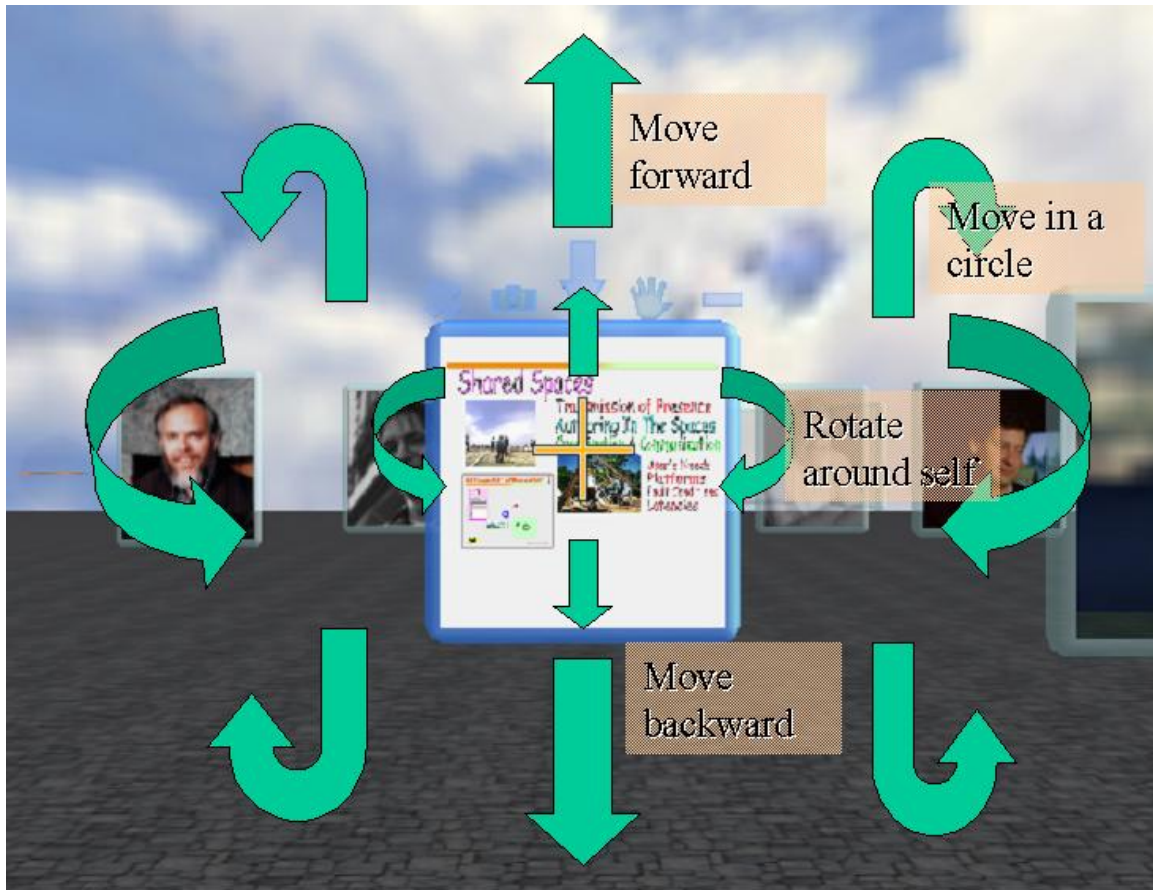


Moving Around

Let's learn how to move around inside of the Croquet world. You should notice an orange cross hair at the center of the 3D screen. We use this to help navigate inside the world. Moving around inside of the 3D world is quite easy, but does take a little practice.

To move around inside the world, just click and hold the right mouse button. Where you click relative to the cross hair determines how you move. The closer you are to the cross hair, the slower you will move. To move forward, move the cursor on top of the cross hair and click. The distance from the center determines your forward velocity. Click and

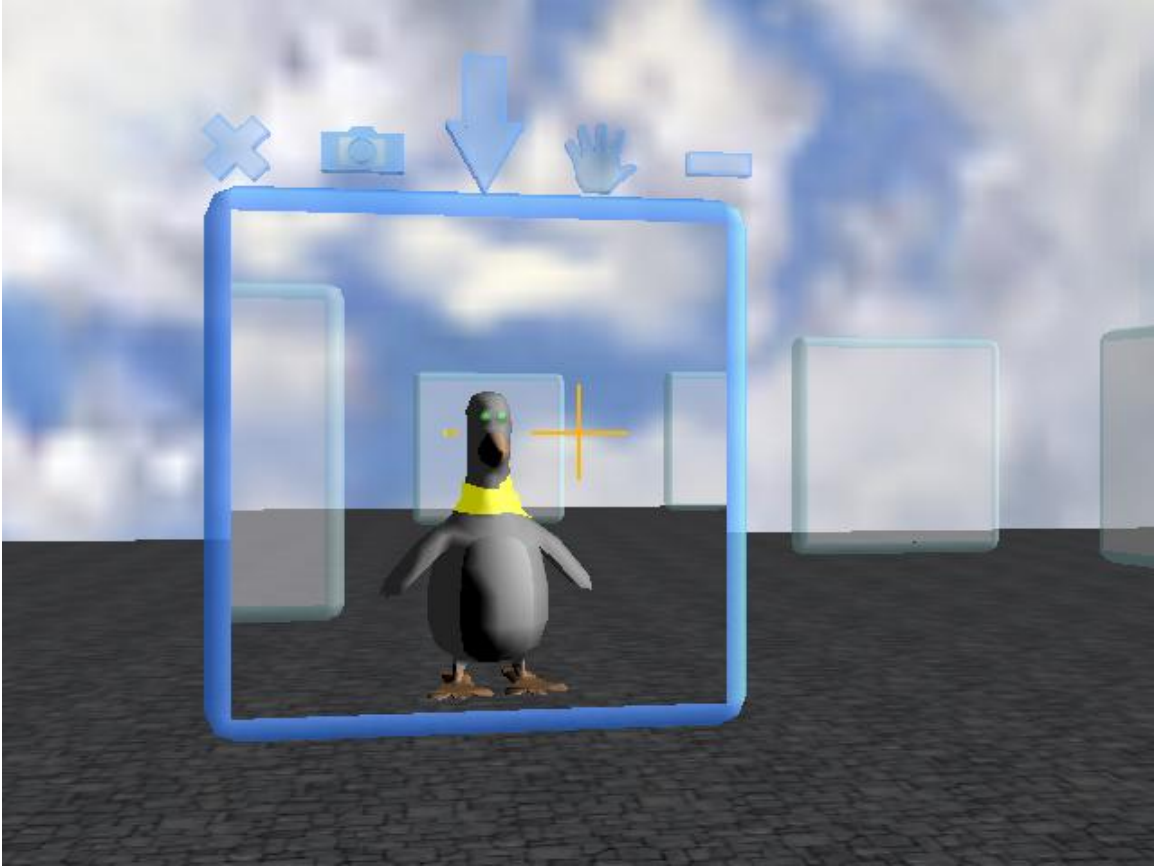
hold just above the cross hair and you will move forward very slowly. Click far from the cross hair and you will move quickly.



If you move the mouse underneath the cross hair, you will move backwards. Moving it right rotates you to the right. Again, distance determines velocity – in this case, angular velocity, or the speed at which you rotate. If you are directly over or under the cross hair you will move in a straight line with no rotation. If you move directly to the left or right of the cross hair, you will rotate around your center without any forward or backward motion. If you put the cursor in just above and to the right of the cross hair you will move forward a bit and rotate to the right a bit – all at the same time. This allows you to walk in a circle.

You can also look up and down by holding the shift key as you click up and down. Instead of moving forward, you will look up. This is spring-loaded, so as soon as you release and begin to walk around again, your view will return to straight ahead.

Now see if you can find the mirror window. Your location in the Croquet space is currently represented by a penguin avatar, so you should see something like this when you find it:



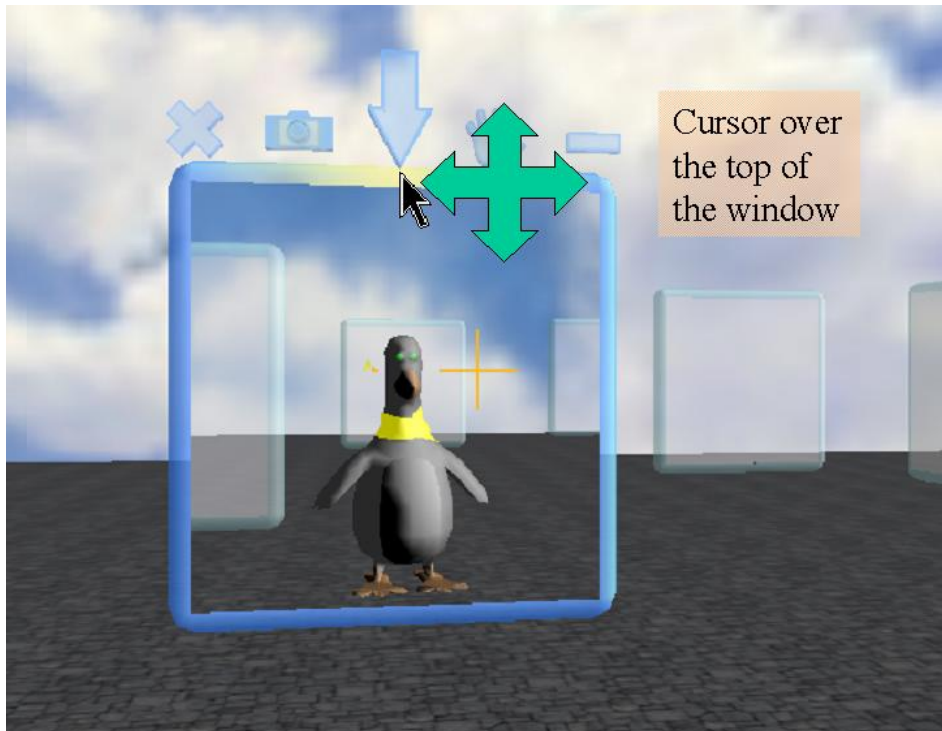
A mirror window. The user is represented by the penguin shown in the mirror.

4. Picking and Manipulating – Croquet Windows

Selecting objects and moving them around is very similar to moving objects around in a 2D world. We will use the 3D windows in Croquet to demonstrate this.

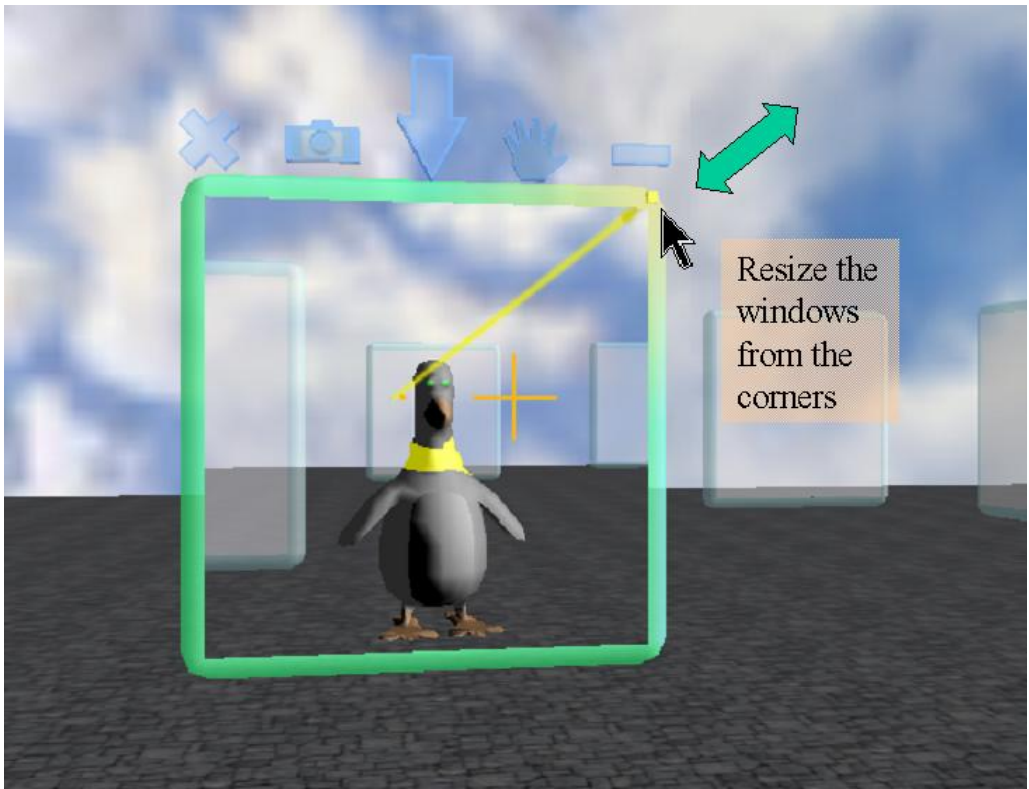
Note: The 3D windowing system that is provided with Croquet was designed to demonstrate how one might use 3D objects to display and manipulate information. Unlike 2D windowing environments, where windows are the central presentation paradigm, in Croquet, windows are just one example of many potential models of interaction.

Let's move up to one of the 3D windows and start playing with it. All of the windows will work the same way, so you can use any one that happens to be handy. The first thing we will do is move a window. To do that, simply grab the window by the top part of the frame. It will glow yellow in the center when the cursor is in the right location.

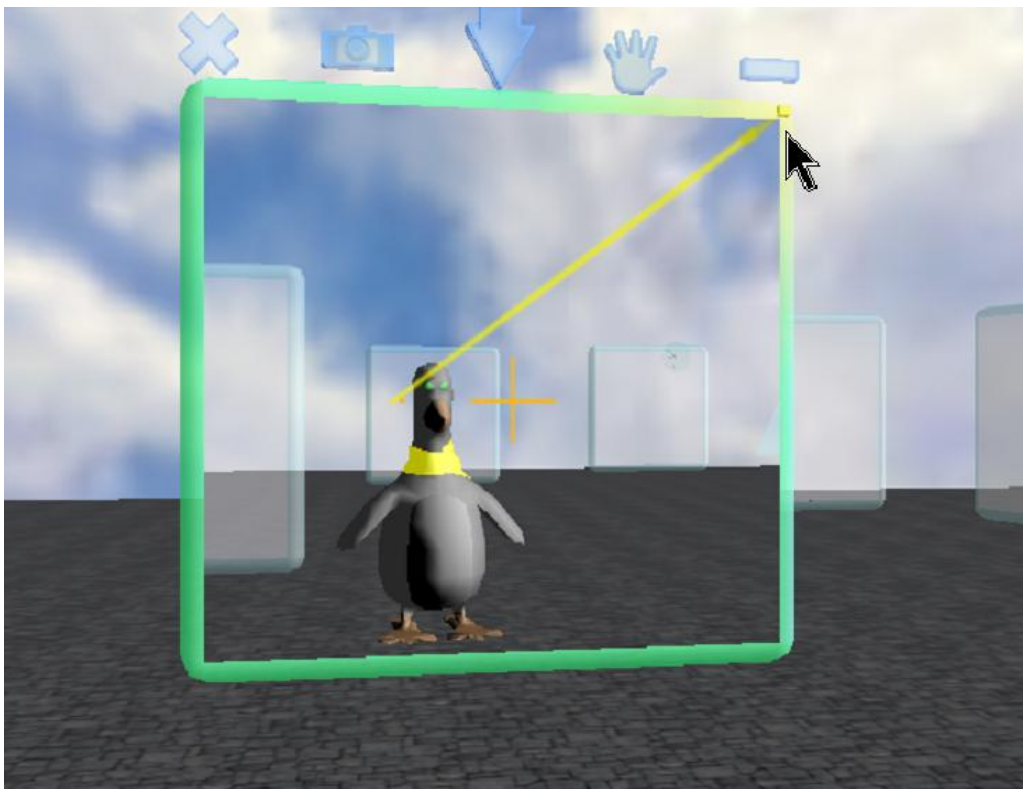


Simply click and drag the window. It will move in a plane that is perpendicular to the direction of you are looking in.

Now move the cursor over one of the corners of the window. Clicking and dragging here allows you to resize the window.

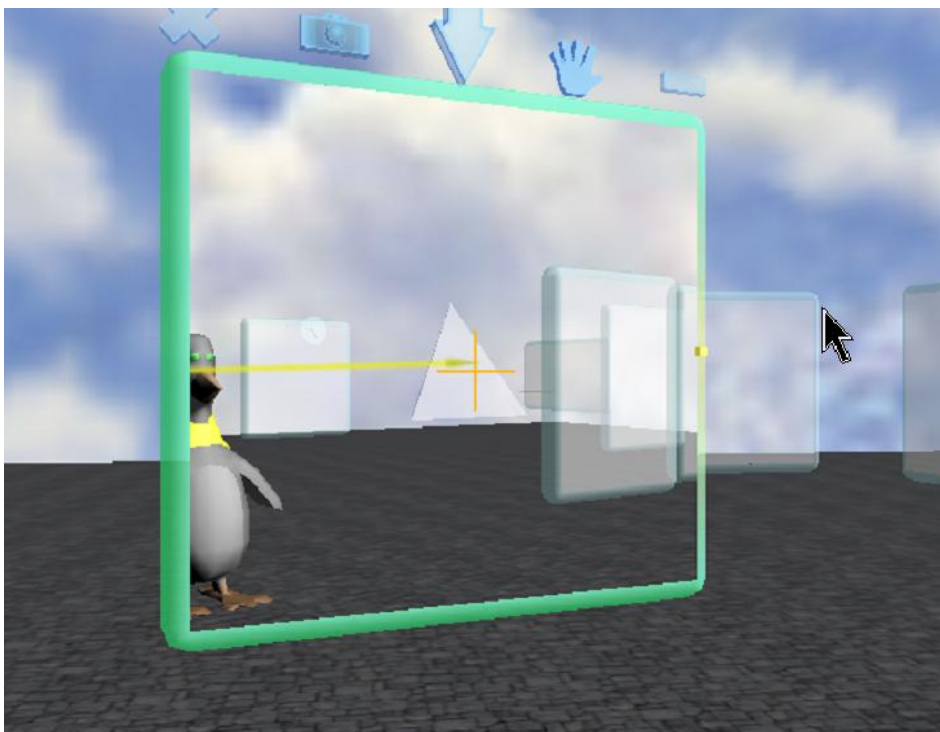
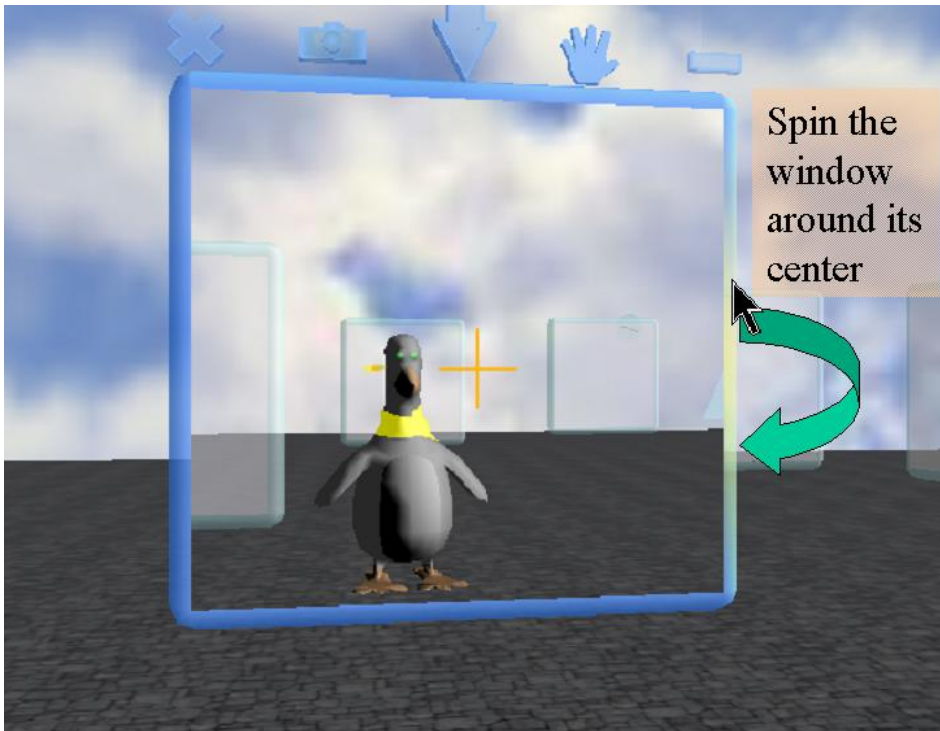


Resize a 3D window from its corners. Any of the four corners will work.



The window is resized. The yellow laser line is your pointer visible in the mirror.

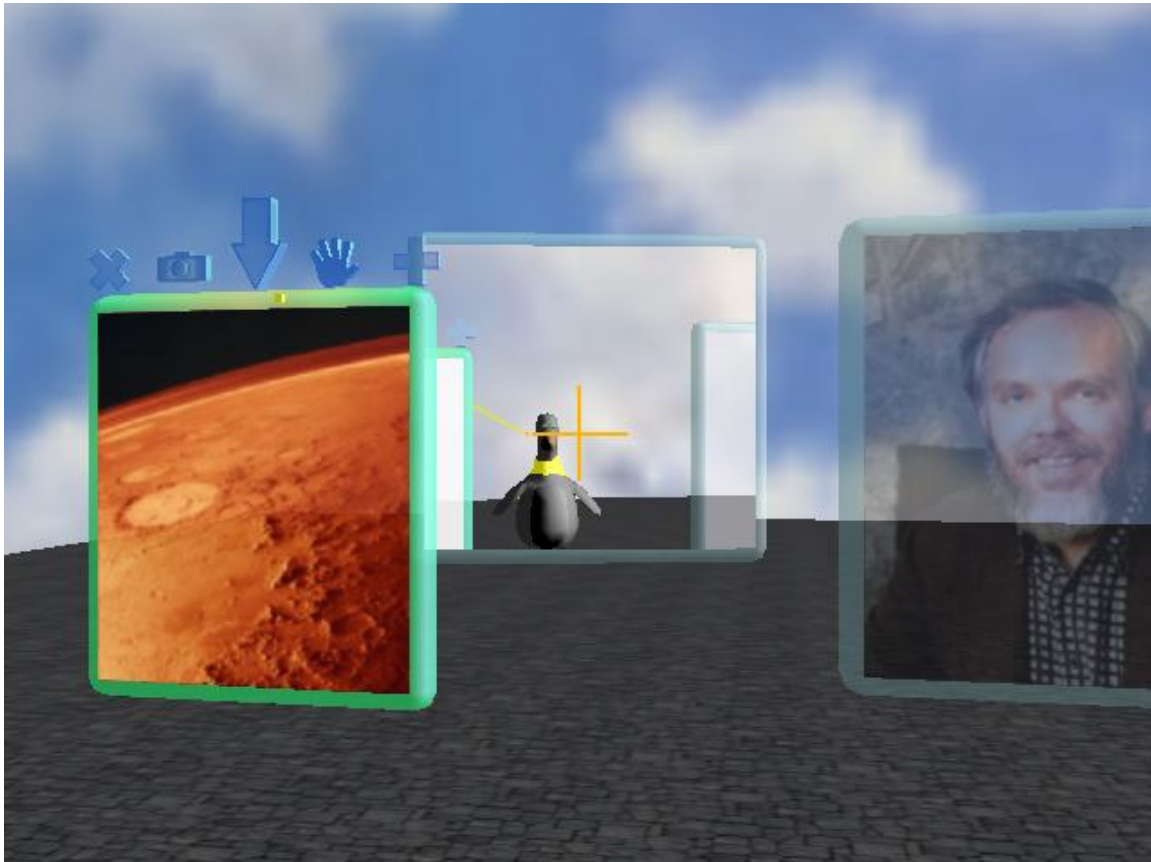
One of the things we can do in a 3D world that we can't do in 2D is actually spin a window. To do that, simply grab hold of either the left or right hand side and drag as shown:



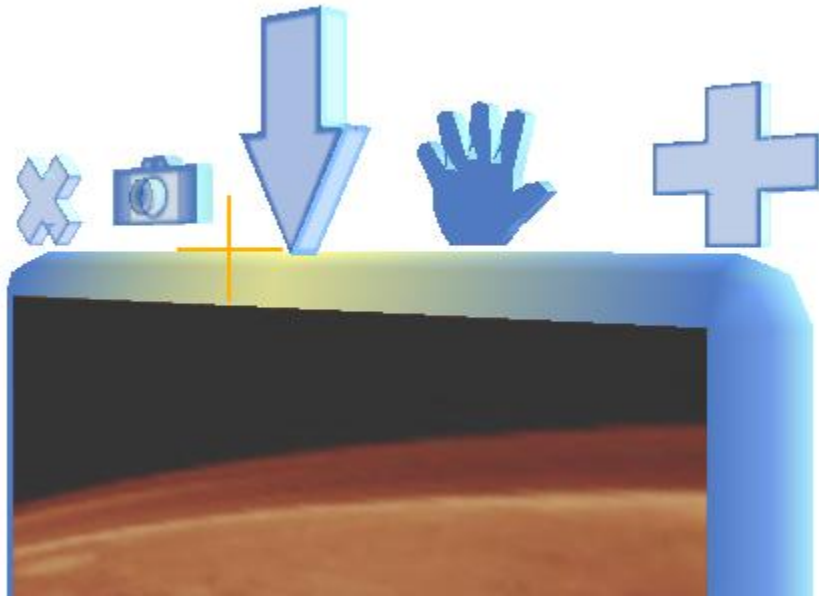
Typically, you will find that the way to pick and manipulate objects in the 3D environment is extremely similar to the way it would work in 2D. There should be very few surprises here.

The Window Halo

Each window has a halo of 3D icons across the top when the cursor is over it. This halo fades when the cursor leaves the window until it disappears. To bring it back, just move the cursor over the top again.



The window to the left has an active halo 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 halo buttons are:



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



The camera button takes a snapshot of the contents of the window if it is a portal. More on snapshots 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 users 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.

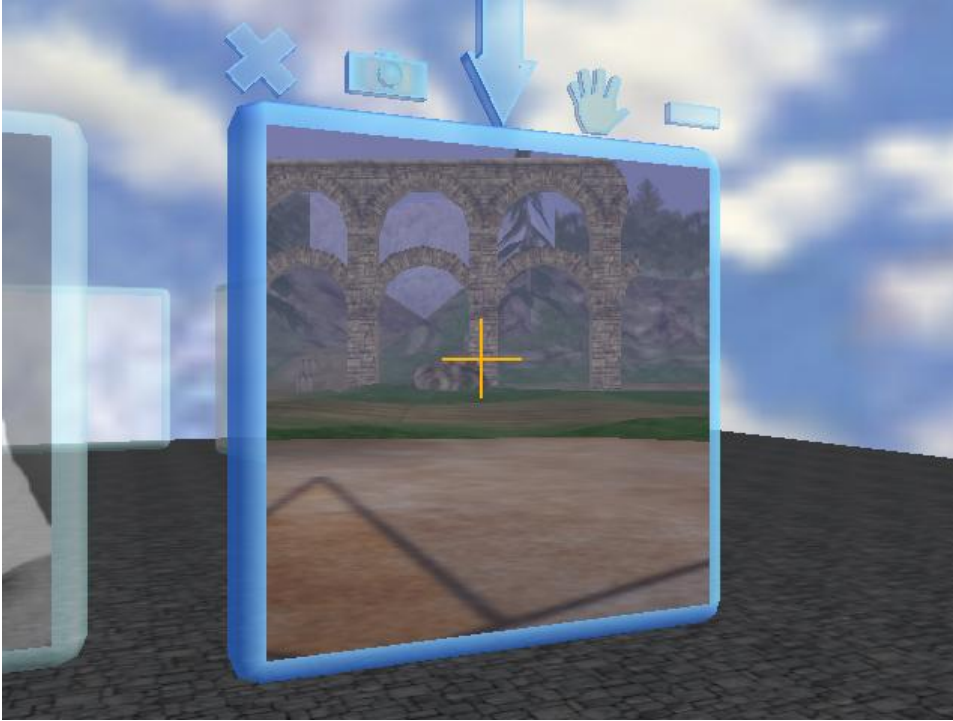
5. Spaces and Portals

Simply put, a Space is a place. In Croquet, a space is a container of objects, including often the user. 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 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.

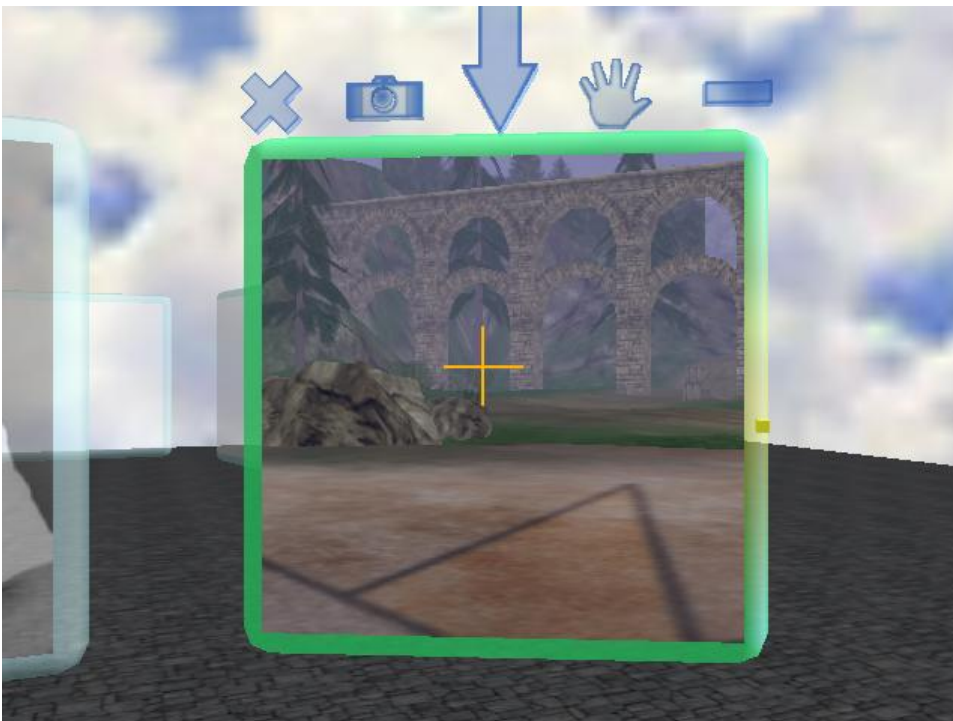
Portals are simply a 3D spatial connection between spaces. If you place one portal in one space, and a second portal in a second space and link them, then you can view from one space into the other. In the example of the child's room, a portal is simply the door to the hallway. The hallway is just another space. One key difference between Croquet portals and spaces and the real world of course 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 it is leaving from.



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



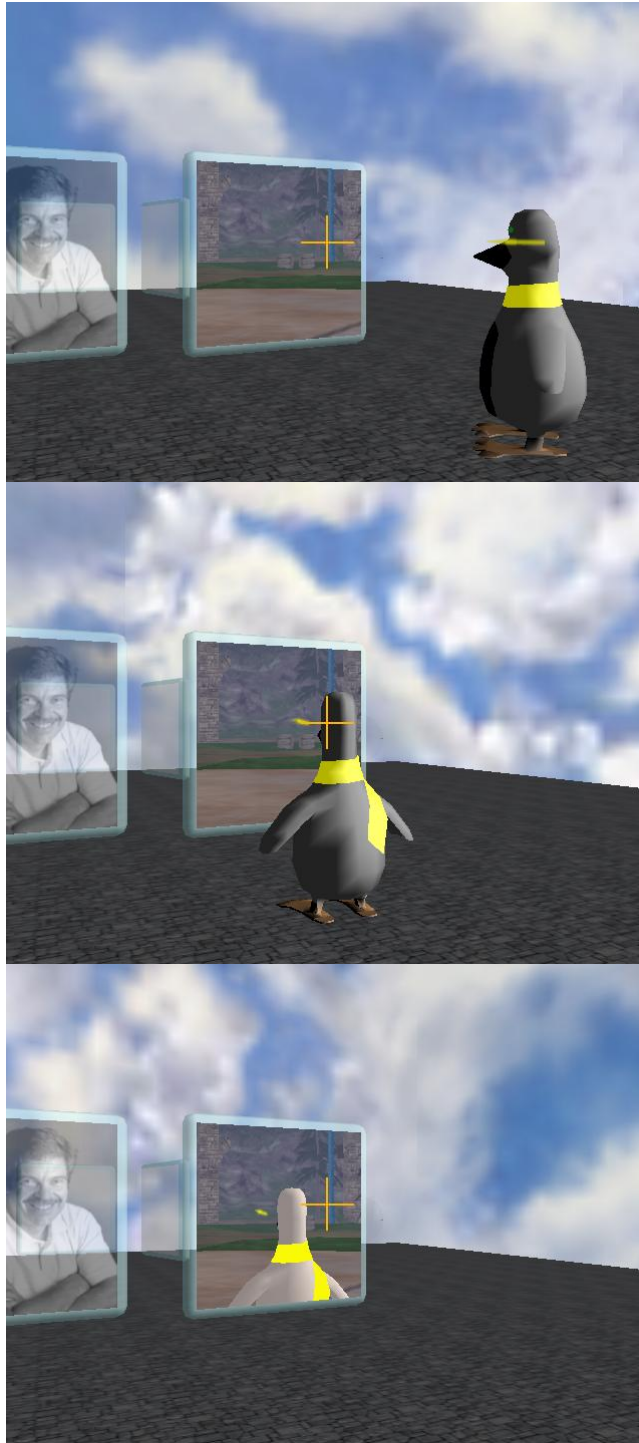
The Portal is now open and we can see into the linked Space, in this case the entrance to a multiplayer game.



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.

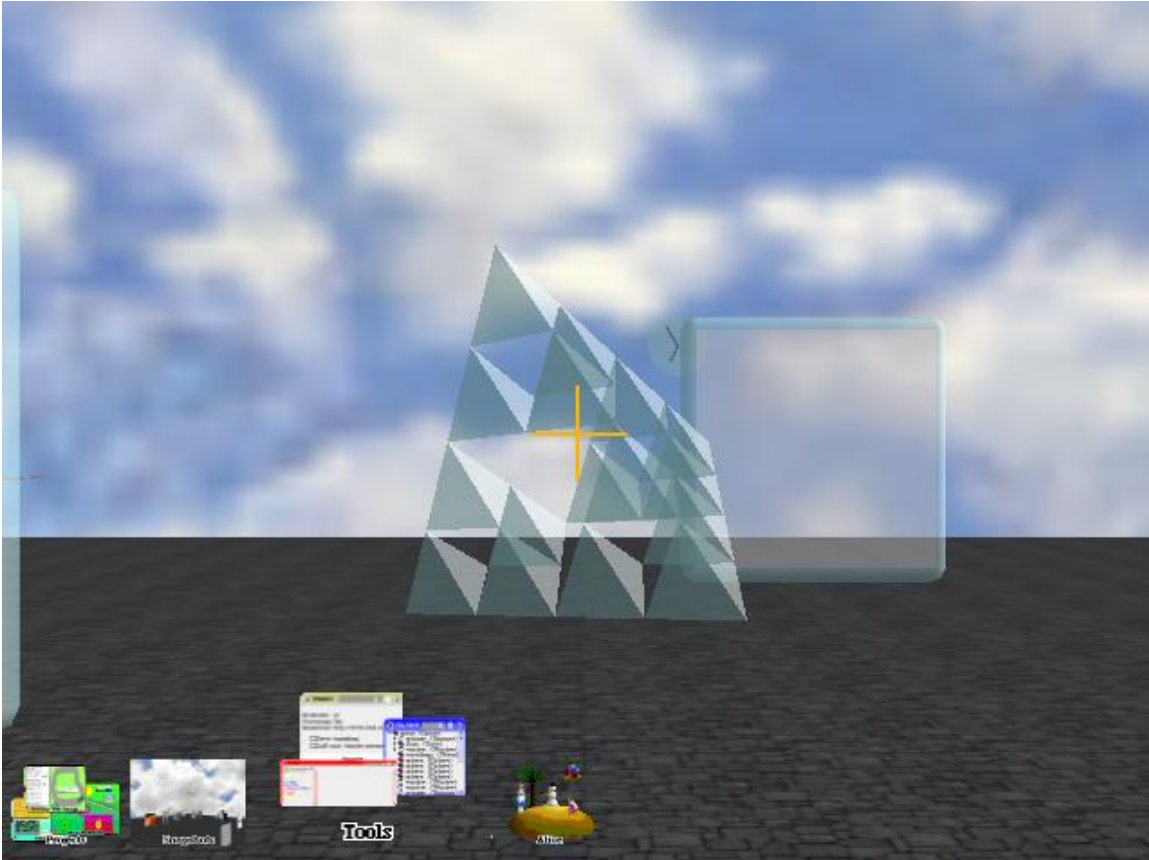
- Portals allow a world designer to partition a space into connected areas; this is now a standard part of any reasonable game engine. If a portal is not visible from

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.

6. Navigator Dock

The Navigator Dock is a place where the user can access 2D Squeak projects, 3D Snapshots of other Spaces, a collection of tools to program and modify the Space, and a collection of models to add to the world.



To retrieve the category of you are interested in, simply click on its icon. This will retrieve yet another list of objects, such as the tools list:



To return to the previous list of items, select the back arrow to the far left.

Projects



In Squeak, a Project is a virtual desktop. Squeak can manage many such desktops at any time, and the user can easily switch between them with their current state being completely saved and later restored.



Select from the list of Projects here.

Croquet allows you to add a 2D Project desktop into your 3D world by simply selecting one of the Projects and dragging it into the current Space.



A Project dragged into a Croquet world is totally accessible.

You can also switch to this other Project completely, simply by clicking on the button. Since Croquet itself runs as a Project, you can restore your 3D world simply by returning to the previous project.

Snapshots

Snapshots are very similar to bookmarks or favorites in a web browser. They are locations that a user has visited and chosen to hold on to if he wishes to come back. Snapshots are made in one of two ways. If it is of a portal through a window, the user simply clicks on the camera button on top of the window and a snapshot of the Space that the portal links to is made and added to the dock.



Another method is to simply press the CTRL-D key, which will generate a snapshot from the current users exact location. Once a Snapshot is created, it is placed into the dock.

Once in the dock, the user can select a snapshot and be immediately teleported to that location.



To retrieve your current Snapshots, select the Snapshots icon in the Navigator Dock. You will see a view much like this one:



Click in the snapshot at the lower right of the screen and you are instantly teleported to:



the world where that snapshot was created.

Notice that if the world that the snapshot is pointing at is already loaded in your system, the snapshot that your cursor is over will begin to display a live image of that world.

Tools



The TPainter tool is particularly interesting, because it allows the user to create 3D objects just by painting them. Click on the TPainter icon and you will see the overlapping painter window. Simply select your tools and draw. Once you are satisfied with your work, press the “Keep” button and an “inflated” version of your drawing will be created and placed into the scene. If you would prefer a billboard or flat version, simply hold down the keyboard Shift button when the press the “Keep” button.

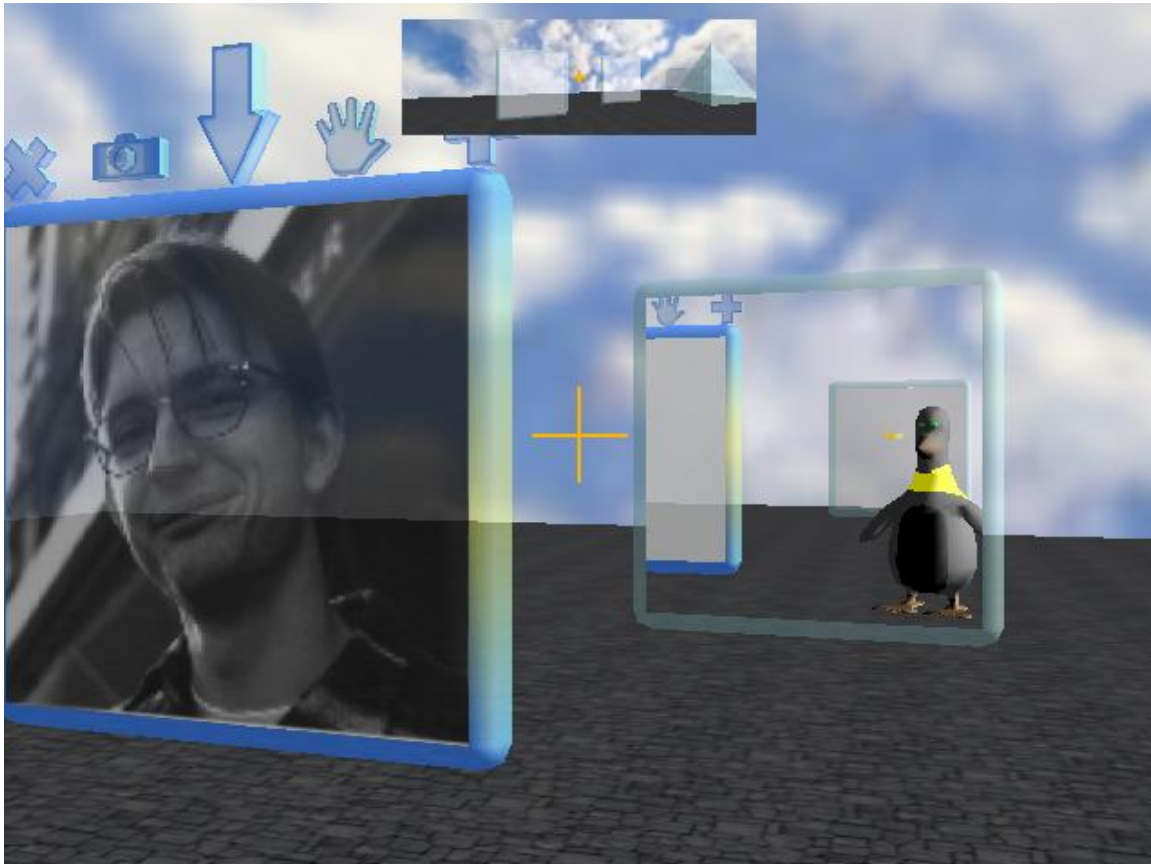


Models

In addition to the other objects and capabilities accessible from the Navigator Dock are a large number of models that can be dragged and dropped into a scene. These objects can then be scripted and animated.

7. Additional Controls

There are many hidden controls inside of Croquet. We are working at making these a little bit more visible, so these will probably be triggered by buttons or other more obvious interfaces. For now, you just have to remember how to get them.



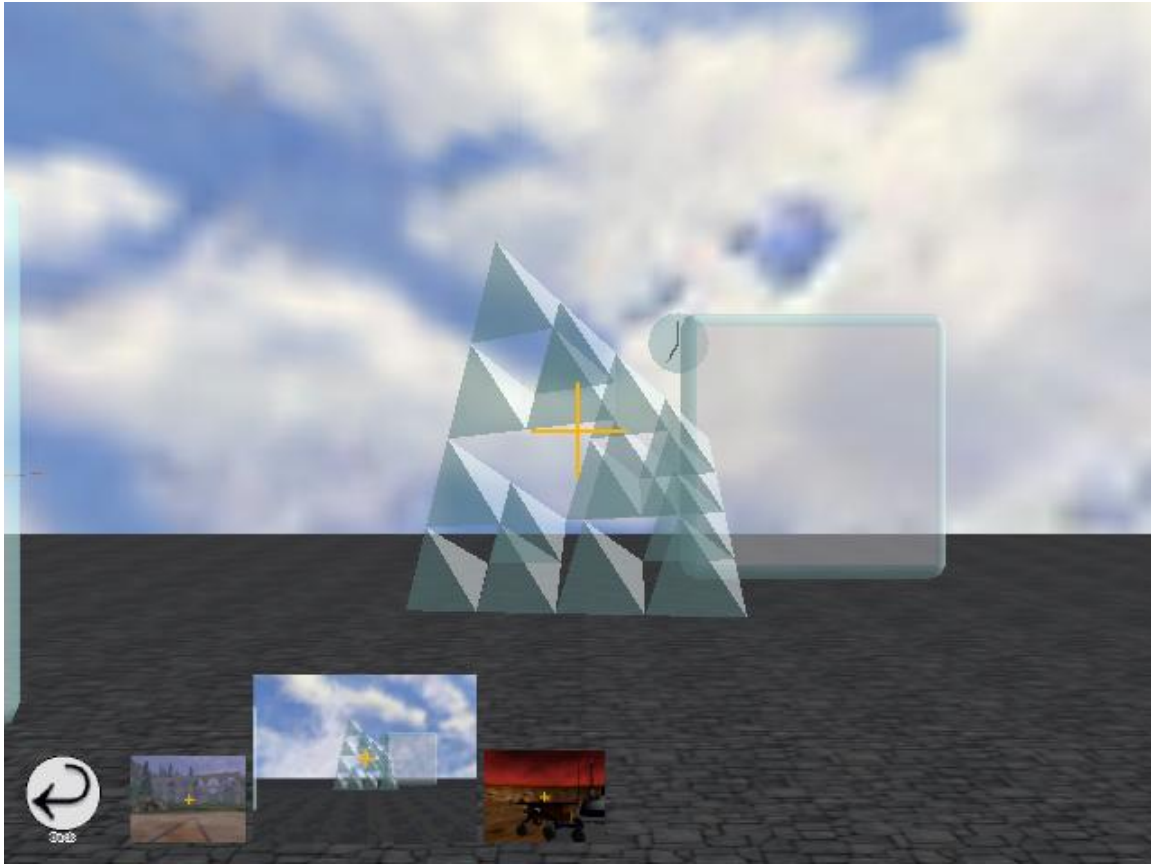
Rear view mirror

The rear view mirror can be instantiated with the default camera simply by pressing CTRL- R. It then appears at the top center of the screen and displays exactly what is behind the user as he moves through the Space.



Overhead view

An overhead view is displayed if the user presses CTRL-O. This is a top down view on the user's avatar and displays the user and everything around him. This appears at the top left of the screen.



Camera snapshot

A snapshot can be made from anywhere in any world simply by pressing CTRL-D (for Dock). Once this is done, a snapshot from the camera's current location appears in the Navigator snapshot dock.

8. Croquet: Peer-to-peer

Croquet was designed as a peer-to-peer operating system from the start. Every decision we made was within this context. But even with this focus, much of the true collaboration capability is incomplete. To use Croquet collaboratively today requires that both you and your peer-to-peer partners have access to a broadband connection to the Internet or all of you should be on the same high-speed LAN if Internet access is not available possibly due to security issues.

Again, when Croquet starts up, you will see something like the following window on your screen. Check your nickname. You can put anything you want here. The partyname should probably be Tea for now, though you are welcome to experiment. The Rendezvous is an Internet location that maintains a list of the current connected users. It is not used for communication. Once you find the other users, you communicate directly with them. “*Force tunneling*” should always be checked, and if you are working on a LAN with no Internet access, also check “*LAN only (disable Internet)*”. Once the checks are properly set, click the “*Connect*” button at the bottom of this window.



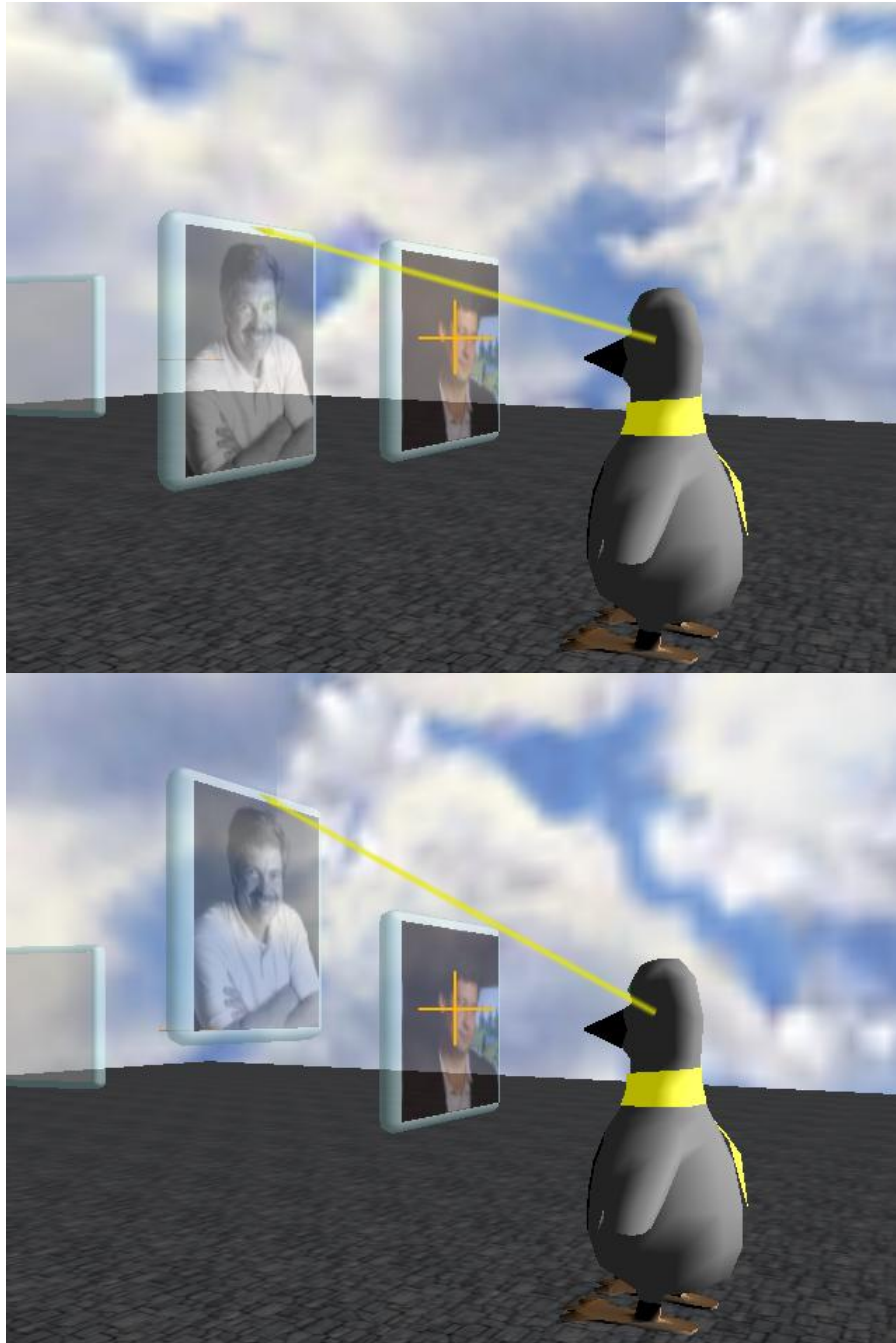
You will now be placed into “conference” with your peers. Some users may notice a significant delay before other users appear and your machine may appear to freeze up for

some period of time. Don't give up, control of your machine will (usually) be returned to you.

Once you have made the connection you will be able to see the other participants in the space. It is very likely that all of you will start up standing in the same place, so you will probably have to move around to see anyone. The current avatars are all penguins. This is a placeholder for individualized models that will be developed. Place your cursor over one of the avatars to see his connection information, including his nickname and computer's IP address.



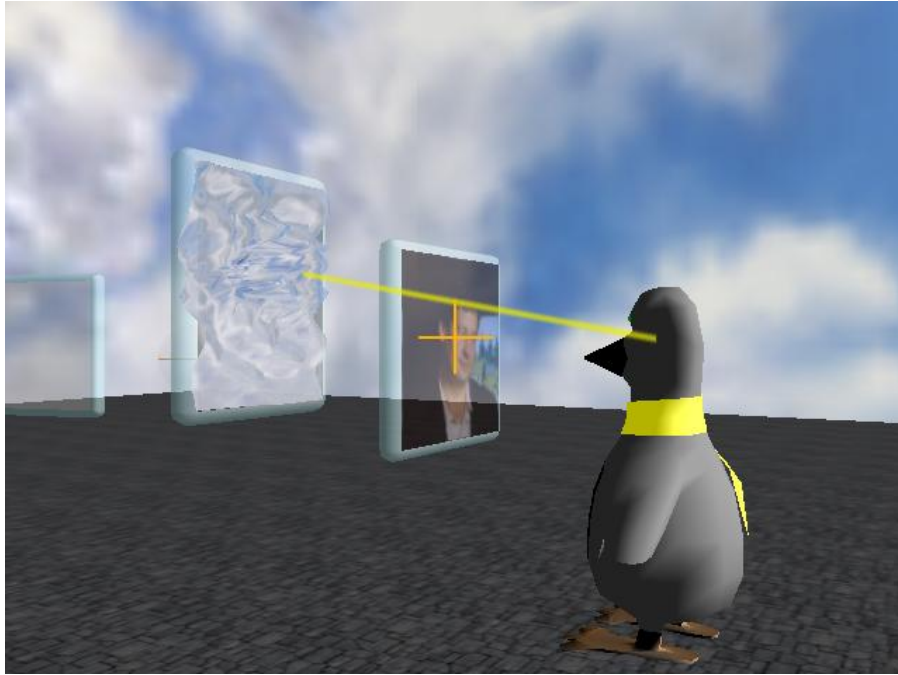
You will probably notice a small laser-like arrow beside the head of the avatar. This is actually the avatar's pointer, and represents where the avatar's user is placing his cursor. When the user selects an object, the arrow extends to touch the selected object so you can see what he is doing and to what. As an example, the user in the images below has selected and is dragging the window.



The other user is dragging a window up into the air.

A remote user can do anything in this shared world that you can. The environment acts as a single shared place that every inhabitant can manipulate and modify. You have a context to see and understand where the other user's interest is and further, what it is they are doing.

Here is an example of the other user now clicking inside the contents of the same window that he was previously dragging.



9. Scripting in Croquet

Exploring spaces

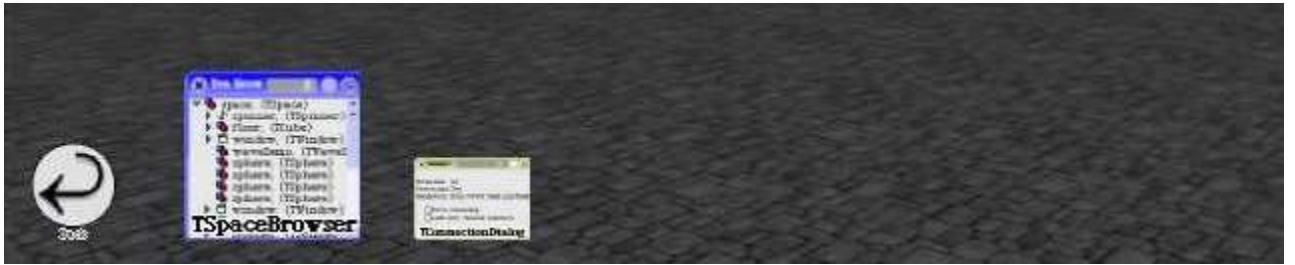
Let's assume we start out in our initial world and want to do some stuff. Your world should look like the following:



Move the mouse to the bottom of the screen to get the navigation dock:

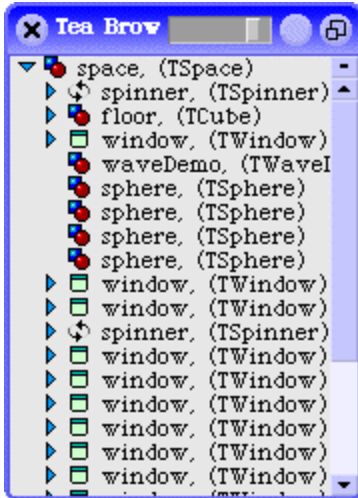


Click on the tools to get to get to the space browser:



Click on the space browser itself. It will pop up in the midst of your screen:

The space browser is a good place for starting to explore the internals of your space. It will show the various objects, scripts, and actions in your world. Let's open the space (click on the little triangle to the left of the icon) to see what's in there:



Lots of stuff eh?! Well let's see what's what. Click on the first window in the list to highlight it and then **RIGHT-Click** (Mac users: That's **OPTION-Click**) to get a context menu. It will look like the following:

There are a number of interesting things you can do in here. Let's start with the simple ones:

*** go near**

This will take you near the object you have selected. It is often useful for finding out what this object actually is if it does have a non-descript name (like "window"). So let's do it. When you click on "go near" you will turn towards the object and move to a relatively close distance. Try it out! Click on a few of the other objects in the list and "go near" them.

*** turn to**

This will only turn you towards the object. Not quite as useful as "go near" but still helpful if you can't remember where this thing was but don't want to leave the place where you are.

[**TODO:** We need something like a "do" or a "try" menu which lists the individual actions (like move, turn) and allows to do them or to just try them out with automatic undo afterwards].

* **destroy <object>**

This command will do exactly what it says - it destroys the object. So be careful!

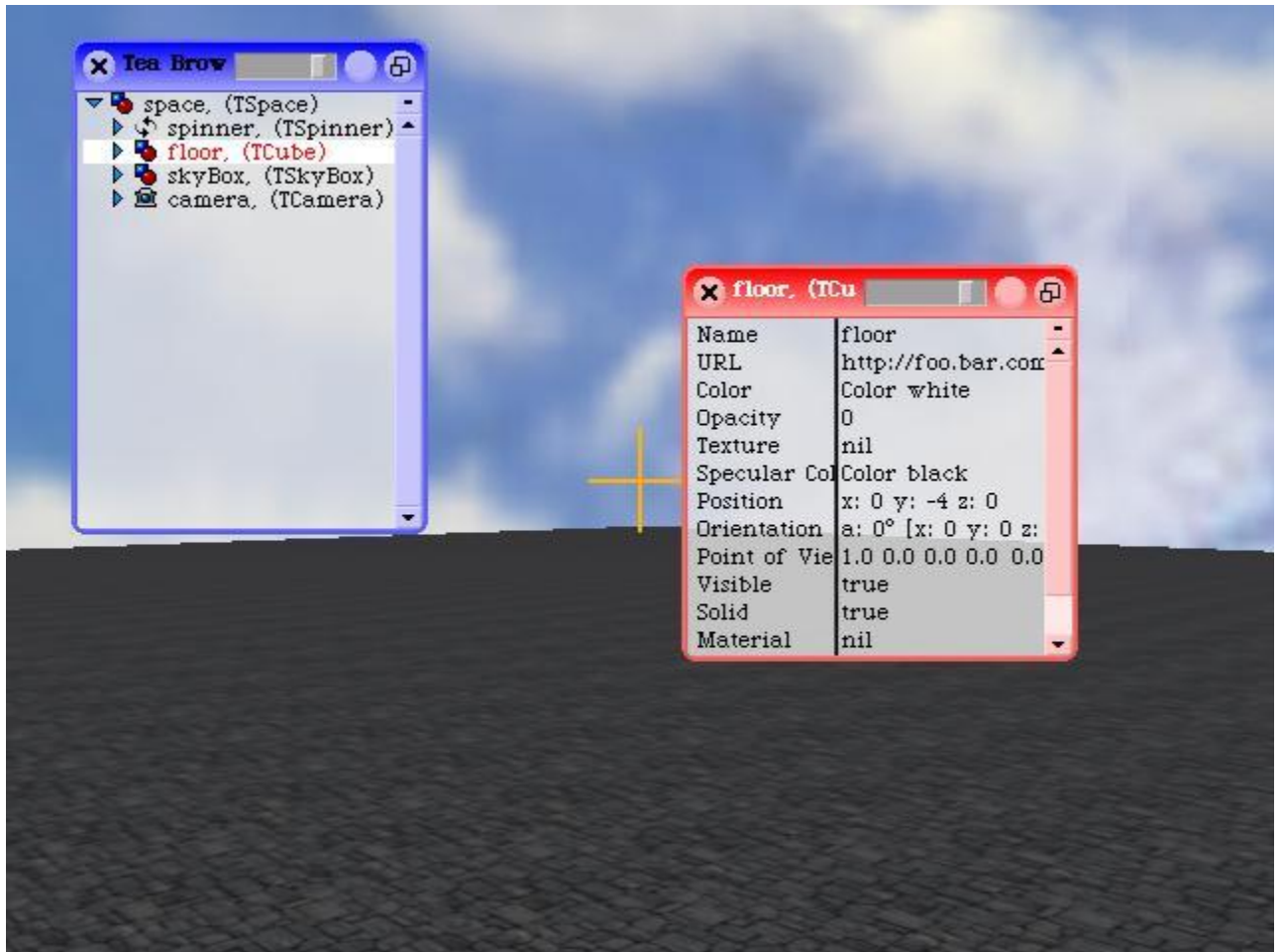
Exercise: There is too much stuff in the world right now. Use the "turn to" "go near" and "destroy" commands to check out all objects and destroy everything but the spinners, the floor, the skybox and the camera.

As a result, your world should look like:

There's still this pyramid in the world and we want to get rid of it too. Let's use the "go near" command to see which one it is and then destroy it. Now we've got a clean world to play with.

Properties of Objects

In our almost empty world we may want to find out a bit more about the individual objects. In the space browser, click on the "floor" object and **DRAG** it into the 3D space. Drop it there. You will get a new red window showing the individual properties of that object:



There are various entries which you can click on and edit:

[WARNING: MOST OF THESE DO NOT WORK AT THIS POINT]

- * **Name:** The name of the object
- * **URL:** The location where this object was loaded from
- * **Color:** The color of the object
- * **Opacity:** The opacity of the object
- * **Texture:** The texture map of the object
- * **Specular Color:** The specular color of the object

- * **Position:** The position of the object in the world
- * **Orientation:** The orientation of the object
- * **Point of View:** Position + Orientation
- * **Visible:** Visibility of the object
- * **Solid:** "Solidity" of the object. "solid" objects you can walk on.
- * **Material:** The material of the object
- * **Children:** The children of the object

These properties (at the point where they'll be working ;-)) will enable you to modify, exchange, and inspect various interesting parts of objects. At this point most of them do **NOT** work.

Note: Close the red window before continuing (e.g., click on the top left button with the "x" in it)

Making scripts

Let's make some of our own scripts. First of all, we need something more to play with. Move the mouse to the bottom of the screen to get the dock. Click on the "back" button



then click on the "Alice" button



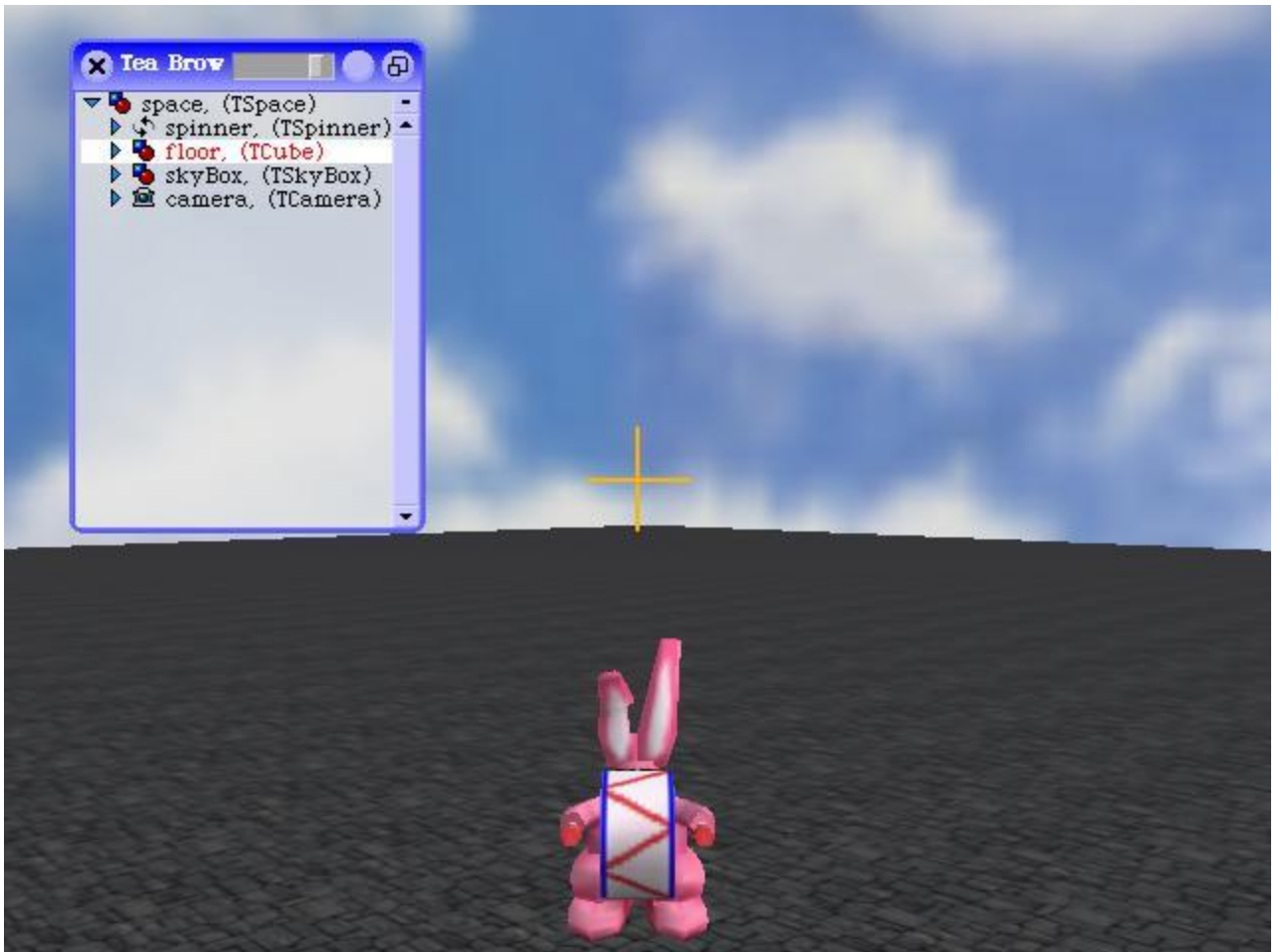
then click on the "Animals" section



then click on "Bunny"

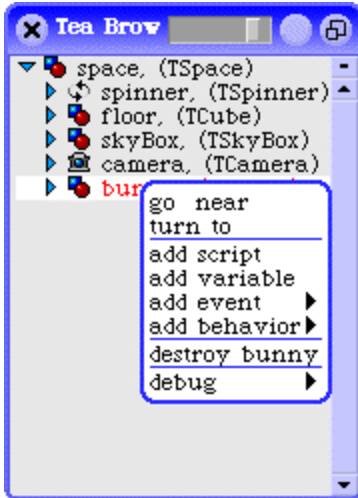


The bunny will pop up in your world:

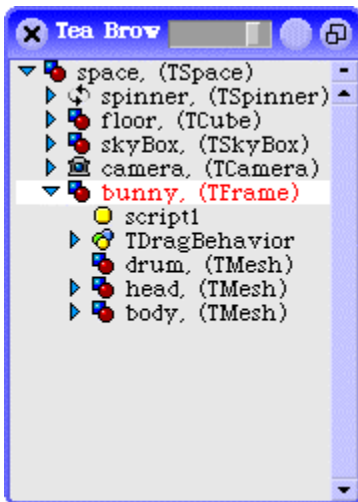


[**BUG:** The space browser will NOT update appropriately. You need to collapse and expand the space (the top-level item) first].

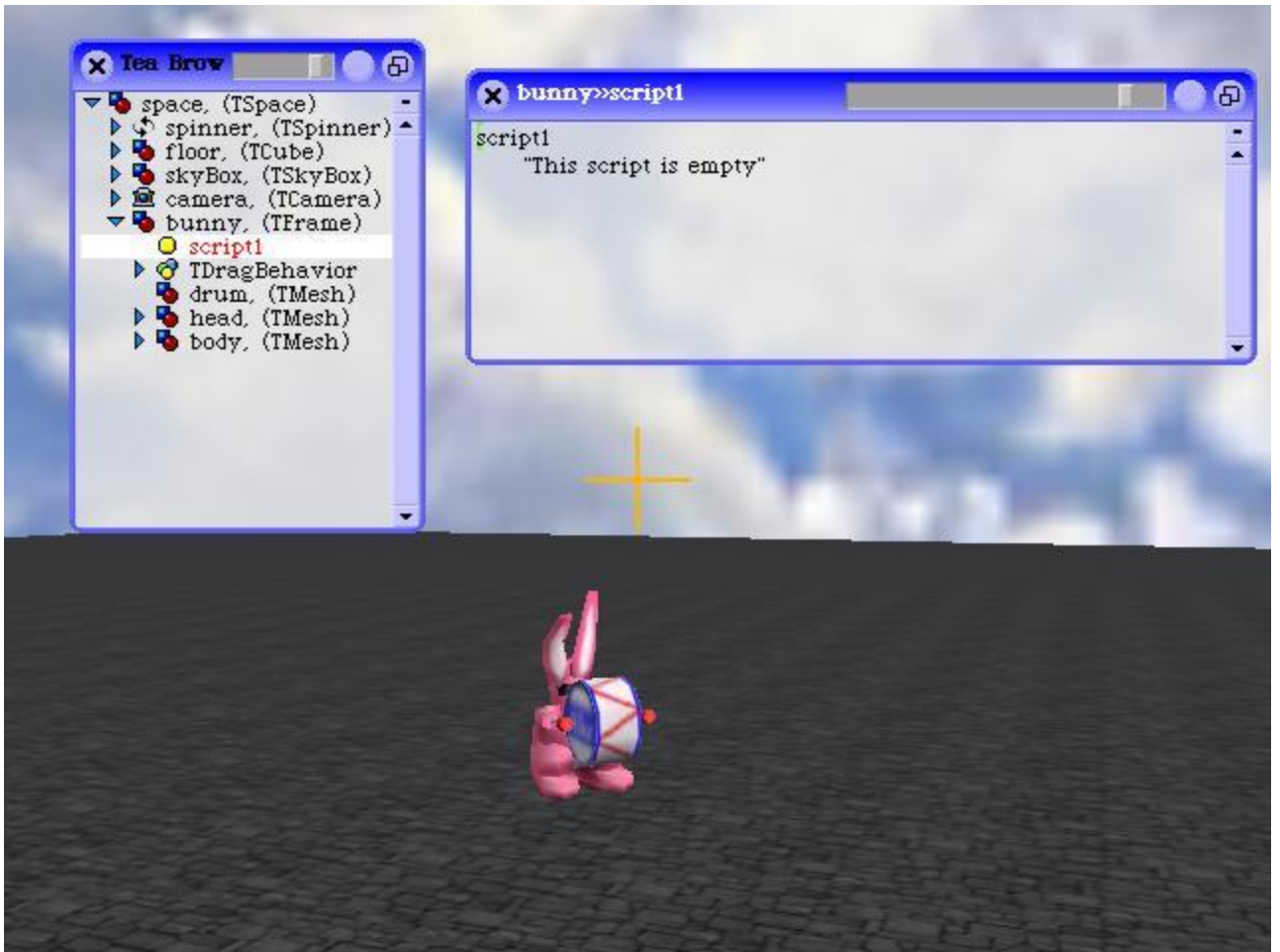
Drag the bunny into a location where you want it to be or drive to someplace else by yourself. Then select the bunny in the space browser and RIGHT-Click (OPTION-Click) to get the context menu:



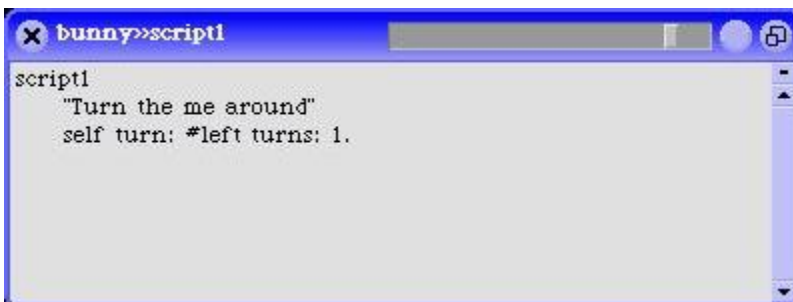
Click on the "add script" command. [**BUG**: This does not automatically expand the bunny if it isn't yet, so you have to click on it for yourself].



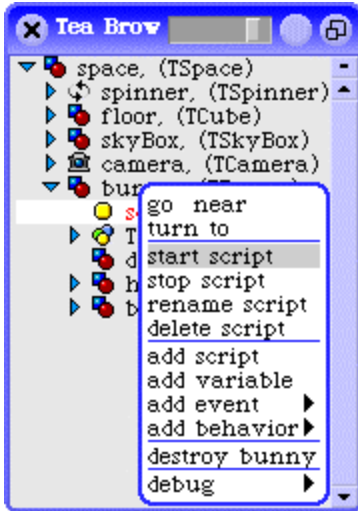
"script1" is the new script that we created. Click on it and **DRAG** it into the world. An editor will show up which contains the code for this script.



Let's make the bunny turn. Write the following:



Then hit Alt-S (Cmd-S on Mac) to accept the script. We can now test the script by clicking on the script in the space browser and choose from its context menu "start script":



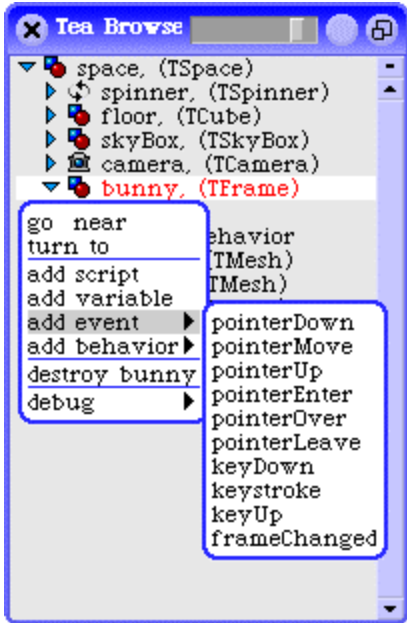
Et voila! The bunny will turn around. Cool, eh?! There are various other commands which you can use (more on this later - this is just going to be a brief tour).

[**TODO:** We need some visual indication about the status of a script, including a time line which shows everything that's currently active].

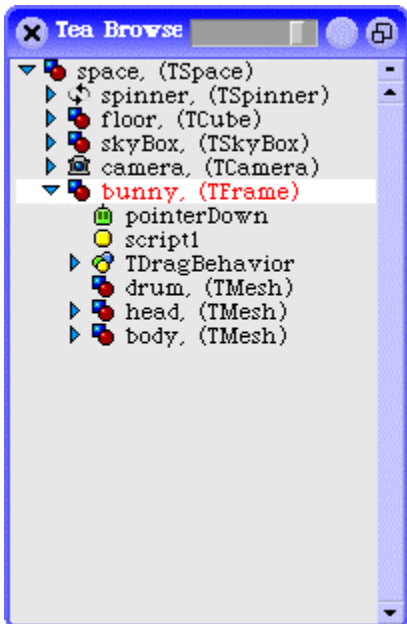
[**TODO:** Say something about "renaming" scripts (if the browser would update accordingly it'd be simple). It does work but it's not obvious].

Making event responses

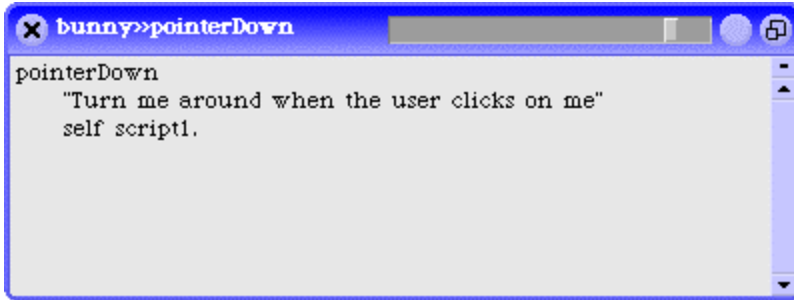
Making the bunny turn when we want it to is kinda fun but it would be even nicer if we could make it turn when we click on it or when some other event happens. For making an event response we need to click on the bunny, then RIGHT-Click (OPTION-Click) to get the context menu and go into the "add event" submenu.



These are the events bunny can react to. Let's stay simple here and just use "pointerDown" [TODO: Think about these names - is "pointerDown" really a worthwhile complication if we could just use "mouseDown" and friends?! Why bother explaining the difference between a mouse and a pointer if we could just make it polymorphic behavior?!].



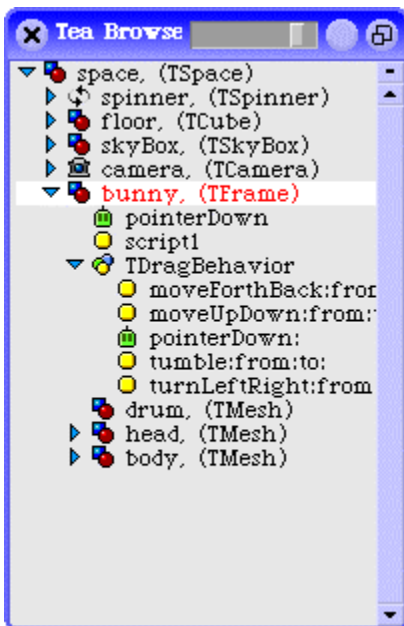
Drag the "pointerDown" script out in the world and edit it so that it reads:



"Accept" the code (e.g., hit Alt-S/Cmd-S) and click on the bunny. It will turn around whenever you click on it. Note that you can still drag the bunny around - it will spin while you drag it.

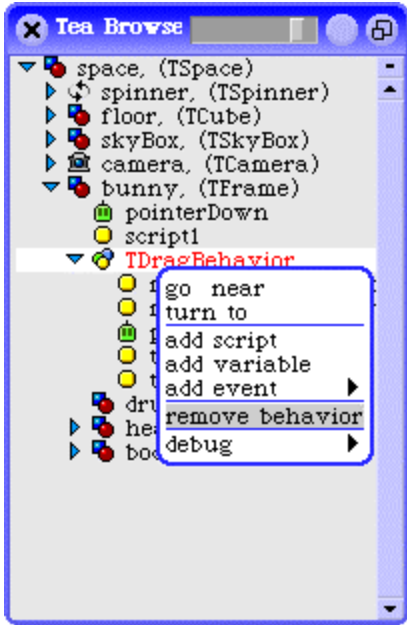
A glance at behaviors

Once you have more complex scripts you might not want to be able to drag the bunny around when you interact with it. The dragging itself is what you see in "TDragBehavior" which - once you expand it - contains a number of scripts and event responses controlling the drag manipulation.



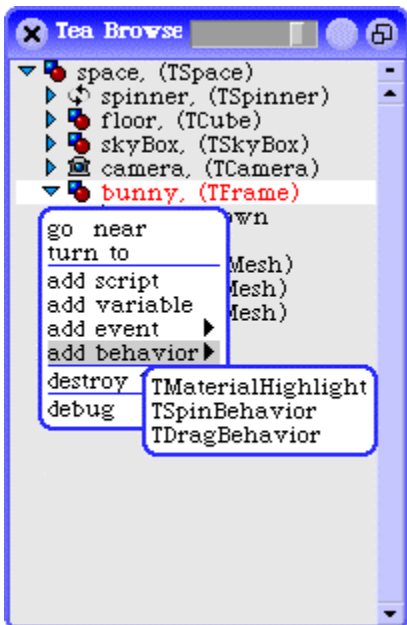
[**Note:** You can drag any of the scripts out in the world to look at them but at this point they are horribly complex]

Let's remove the drag behavior. Click on TDragBehavior, then RIGHT-Click (OPTION-Click) to get the context menu and choose "remove behavior" from it.



It will vanish in a puff of logic. Now click on the bunny. It will still turn (since this is what you told it to do in its pointerDown event response) but you will no longer be able to drag it around.

To get it back, click on the bunny, then RIGHT-Click (OPTION-Click) to get the context menu and from the "add behavior" submenu select "TDragBehavior".

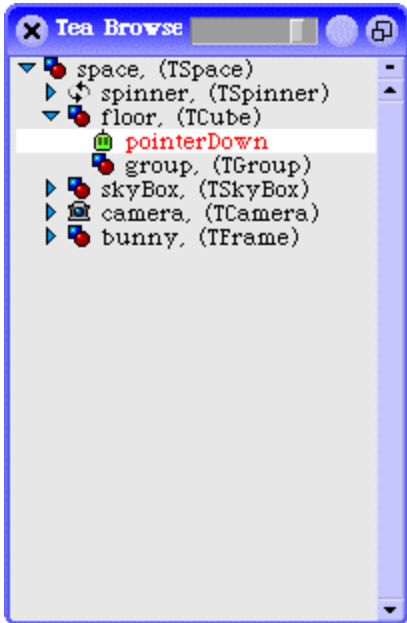


Now you can move the bunny around again.

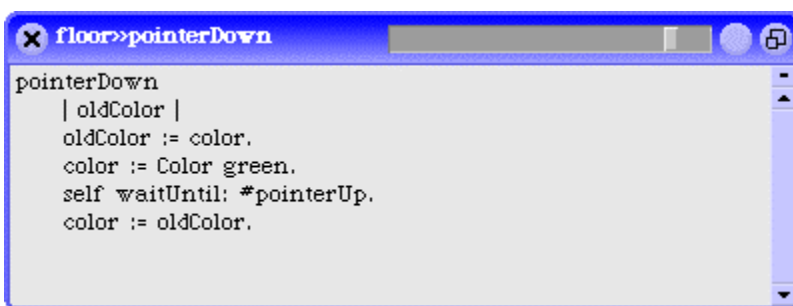
Behaviors are different from "normal" scripts. All of the methods of behaviors are shared between all users of that behavior. When you change one you change all of them. [TODO: There needs to be a way which allows the users to control sharing - we might want to start out with an existing behavior and modify it locally].

More fun with events

There's a lot of interesting things we can do with events. Let's do something interesting. Click on the "floor" and then add a "pointerDown" event response.



Now let's write the following script:



Now click on the floor [**BUG BUG BUG: This is horribly broken.** What should be happening is that the floor changes its color while you click on it and restores it afterwards].

The "color" property is what you saw already in the properties of the objects - all of those properties are available for use in your scripts. The names of the properties are the same as in the properties just with their individual words combined to form a single word, e.g.,

"Position" becomes "position", "Specular Color" becomes "specularColor", and "Point of View" becomes "pointOfView".

[**TODO:** I need to think about the property setters more - should they be animated when they're used like in the above?]

[**TODO:** Say more about name spaces. The above is entirely self-referential and that's pretty unlikely use. Should perhaps use explicit references for most examples?].

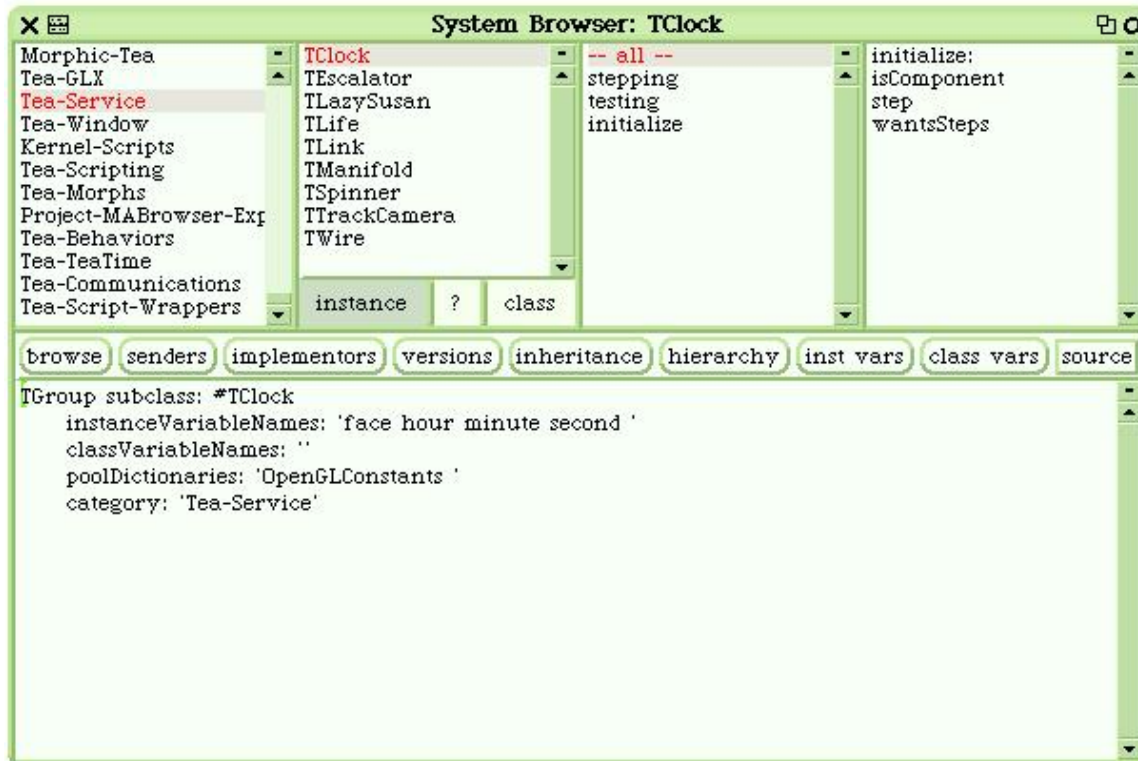
10. An Introduction to Programming with Croquet

To get us started in understanding the Croquet architecture, let's just build something. In this case, we will create a clock 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 clock is a very simple kind of graphical component. It is made up of five pieces – the clock face, the three hands (hour, minute, second) and a base frame containing these objects.

So to start, let's create a new subclass of TGroup called TClock. Below is a picture of what you will see if you launch a Squeak code browser and find the TClock class. Again, I am assuming that you already are familiar with Squeak, so I won't touch on how to get to this window, or how to edit and compile your code.



The first thing to note is that the #TClock class is a subclass of #TGroup, which in turn is a subclass of #TFrame. A more detailed description of #TGroup and #TFrame will follow, but for now, the most important thing to be aware of is that a #TFrame includes a position and orientation in the 3D world relative to its parent #TFrame. Also, I will tend to refer to an object instance of the #TFrame class as a frame. We typically call this position and orientation of the frame a “pose” (a term from robotics).

The pose of the frame is always specified relative to its parent frame. A good example of this is a cow standing in the back of a flat-bed truck. The cow’s position relative to the truck is simple. It is wherever she is standing on the truck bed. It doesn’t matter if the truck is moving or not – her position relative to the truck doesn’t change unless she decides to move somewhere else on the truck bed. Of course, the truck has a relative position to the world, and as long as the road it is on isn’t moving anywhere, we can use this as our global frame. The cow’s global position is simply the position of the truck in the world plus the position of the cow on the truck. If the truck is five miles from Los Angeles (driving away from it – orientation matters), and the cow is three feet in front of the center of the truck, then the cow is actually five miles and three feet away from LA (if LA were a point...)

In the case of our clock, we will be defining the frame of the clock itself relative to the world we will be placing it in, but we will be defining the frames of the face, and the three hands of the clock relative to the frame of the clock. You will see why that is extremely useful way to describe things.

The second thing to notice about the definition of the #TClock is the four instance variable names: face, hour, minute, second. These will be the names of the visible parts of the clock that we will define in the initialization of the object. Let's examine that code now. Again, I am assuming you already know the Squeak variant of Smalltalk.

```
TClock >> #initialize: glX
  | mat |
  super initialize: glX.

  face := TCylinder initialize: glX.
  face topRadius: 0.5.
  face baseRadius: 0.6.
  face capped: true.
  face height: 0.1.
  face slices: 24.
  face rotationAroundX: 90.0.
  self addChild: face.

  mat := TMaterial initialize: glX.
  mat ambientColor: #(0.5 0.7 0.7 0.5) asFloatArray.
  mat diffuseColor: #(0.5 0.7 0.7 0.5) asFloatArray.
  face material: mat.

  hour := TCube initialize: glX.
  hour scale: (B3DVector3 x:0.05 y: 0.35 z: 0.01).
  hour location: (B3DVector3 x: 0.0 y:0.16 z:0.12).
  self addChild: hour.

  mat := TMaterial initialize: glX.
  mat ambientColor: #(0.02 0.04 0.04 1.0) asFloatArray.
  mat diffuseColor: #(0.02 0.04 0.04 1.0) asFloatArray.
  hour material: mat.

  minute := TCube initialize: glX.
  minute scale: (B3DVector3 x:0.05 y: 0.47 z: 0.01).
  minute location: (B3DVector3 x: 0.0 y:0.22 z:0.11).
  self addChild: minute.
  minute material: mat.

  second := TCube initialize: glX.
  second scale: (B3DVector3 x:0.02 y: 0.5 z: 0.01).
  second location: (B3DVector3 x: 0.0 y:0.23 z:0.13).
  self addChild: second.

  mat := TMaterial initialize: glX.
  mat ambientColor: #(1.0 0.05 0.05 1.0) asFloatArray.
  mat diffuseColor: #(1.0 0.05 0.05 1.0) asFloatArray.
  second material: mat.
```

So, let's start from the top.

```
TClock >> #initialize: glX
  | mat |
  super initialize: glX.
```

The first thing we do in the initialization of the #TClock is ensure that the superclass #TFrame itself is properly initialized. This requires that we call #initialize: with the glX object as its argument. The local instance variable name of the glX object is glx, so I will refer to that.

This glx object holds the interface to OpenGL, as well as managing the texture directories and a few other chores. Croquet uses the OpenGL engine to perform all of its rendering, and each individual frame has full responsibility for actually making the proper OpenGL calls to accomplish this. Don't be too concerned, however. Most people will probably never need to actually learn OpenGL, as there is already a very rich collection of objects that can perform most of the tasks required. On the other hand, the entire OpenGL interface is readily available so that you can create any effect or extend the system in any way you like.

We also defined a local variable here called mat. This is short for material, and we will be using that in just a bit. Now, let's actually make something we can see.

```
face := TCylinder initialize: glx.
face topRadius: 0.5.
face baseRadius: 0.6.
face capped: true.
face height: 0.1.
face slices: 24.
face rotationAroundX: 90.0.
self addChild: face.
```

Here, we are defining our first instance variable, face. This variable is being defined as a #TCylinder. Notice that the initialization of the #TCylinder also requires the glx object. This is because it is also another kind of #TFrame. Our definition of a cylinder is close to that of OpenGL in that the sides need not be parallel. For example, to create a pointed cone, we can define the radius of the top of the cylinder to be 0. In the case of our clock, we define the top radius to be 0.5 and the bottom to be slightly larger at 0.6. **[There is no real unit of measurement in Croquet, though we will probably settle on a convention of meters or centimeters for consistency.]** We then specify that our cylinder be capped, which is equivalent to putting a lid on top. Then we specify the height of the cylinder to be 0.1 units. The height is obviously the distance from the bottom to the top of the cylinder. The last visual control is to define how many slices make up our clock face going around the edges. The only thing that we ever really draw in OpenGL is flat triangles, so to get the appearance of a curved object, like our round clock face, requires that we break the curve into lots of smaller flat triangles. In this case, we slice the edges up into 24 pieces going around the edge. Fewer slices, and you would see that the clock face really isn't all that round at all.

This creates the clock face that looks something like a bottle cap. Unfortunately, it is created so that the clock face is facing straight up in the air. To make it face us, we need to rotate it 90 degrees around the x-axis.

The last thing we do is add the face to the #TClock's frame, just as we had the cow in the truck's frame earlier. In this case, you can't actually see the #TClock object, as there is simply nothing to see. It is just a location and orientation. What you will see instead are the pieces that are contained in the #TClock as child frames.

Now that we have created the clock face, we want to add a bit of color to it. To do this, we have to create a material object with which to "paint" it with.

```
mat := TMaterial initialize: glx.  
mat ambientColor: #(0.5 0.7 0.7 0.5) asFloatArray.  
mat diffuseColor: #(0.5 0.7 0.7 0.5) asFloatArray.  
face material: mat.
```

We are finally using our local variable name, `mat`, and creating and initializing #TMaterial object. Notice that AGAIN we are using the `glx` object in the initialization of the object. The reason we do this is that the #TMaterial class is also a subclass of #TFrame. There are a number of reasons for doing this. One is that a #TMaterial object can actually be rendered. It actually looks like a teapot (surprise!) We do this if we ever need to experiment with materials independent of what the material will be applied to. Another reason is that materials themselves must make OpenGL calls to set themselves up when they are being painted onto another object, and to restore the world back to the original state when they are done.

We are defining two colors here – the ambient and diffuse colors of the material. In this case they are being set with the same values. You can find out what the meaning of these material values are in any OpenGL reference. We are setting these values to be a four element FloatArray object. The four elements of this array are the red, green, and blue color values (in that order) and the alpha, or transparency value. In this case, we have high values for blue and green, with a smaller amount of red mixed in. Also, we are specifying that it be about 50% transparent. This results in the very nice light blue-green color of the clock in the early image.

Once we have created the material, we can add it to the face object to be applied when the face renders itself with:

```
face material: mat.
```

The next three objects are the hour, minute, and second hands. These are defined similarly:

```
hour := TCube initialize: glx.  
hour scale: (B3DVector3 x:0.05 y: 0.35 z: 0.01).  
hour location: (B3DVector3 x: 0.0 y:0.16 z:0.12).  
self addChild: hour.
```

The hour hand is defined a #TCube. Again, not the glx object. A TCube is simply a regular 6 sided object – though it need not really be a cube. In fact, the first thing we do here is scale the hour hand with the B3DVector3. This creates a long thin object, which is precisely what we need for the hour hand. The only problem is that it will rotate around its center, which won't be much use to us in telling the time. We need to offset the center, and to do that, we specify a location for the hand that is near one of the ends. Finally, we add it to the #TClock object as well.

The other two hands and materials are defined in similar fashion.

What we have now is a clock that has all of its hands pointing straight up and they aren't moving. The next few methods define the behavior of the clock. The first of these is a test message.

```
TClock >> #isComponent  
  
^ true.
```

That's it. All we are doing here is telling Croquet that this particular object exhibits behaviors – which includes time based steps and event handling. In the case of our clock, there are no events to worry about – just keeping track of the time. To do that, we have to let Croquet know that we want to have time steps sent our way. That is done with this test message.

```
TClock >> #wantsSteps  
  
^ true.
```

That's it. Now Croquet will begin calling the TClock >> #step method defined here.

```
TClock >> #step  
  
| pHour pMin pSec time |  
time :=Time now.  
pHour := -360.0 * (time hours + (time minutes/60.0))/12.0.  
hour rotationAroundZ: pHour.  
pMin := -360.0 * time minutes / 60.0.  
minute rotationAroundZ: pMin.  
pSec := -360.0 * time seconds / 60.0.  
second rotationAroundZ: pSec.
```

**** The #step method is being replaced with the #stepAt: method.

What the `#step` method does is quite simple. First it reads the time now via the `Time` class. Then it converts the hours, minutes, and seconds into the proper angles between 0 and 360 degrees. Once it has these values, it rotates the proper frames by that amount. The hour is rotated around the z-axis with:

```
hour rotationAroundZ: pHour.
```

We are almost done. We have created a clock, but the very last thing to do is to actually drop it into a world so that we can see it. To do this, we call the following method:

```
TeapotMorph >>#makeClock: space
| tframe |

tframe := TClock initialize: glx.
tframe translationX: 10 y: 2 z: 20.
space addChild: tframe.
```

The `TeapotMorph` is a kind of `Squeak Morphic` object. It is responsible for setting up the OpenGL environment and for vectoring events to the Croquet engine. [[The TeapotMorph will be replaced with a more robust and simpler method for creating worlds.](#)] The more interesting object is the `space` argument to the `#makeClock` message. This is a root `TFrame` called a `TSpace`, essentially an ultimate container of other frames. There can be any number of spaces in a Croquet system, but the space always defines the global frame of reference for all objects that are contained within it. Additional duties of a space is managing the lists of lights, portals, and alpha objects that are rendered inside of it.

The only thing that the `#makeClock:` method is going to do is create the `TClock` with:

```
tframe := TClock initialize: glx.
```

Notice again that we are initializing the `TClock` object with the `glx`.

We then move the clock to a nice location inside of the containing `TSpace` with:

```
tframe translationX: 10 y: 2 z: 20.
```

And then add it to the `TSpace` so that it can be rendered and the `glx` object can give it steps to animate the clock hands:

```
space addChild: tframe.
```

And then, when we next run the `TeapotMorph`, we have our clock floating in space. That's all there is to it.

11. TeaTime: A scalable real-time multi-user architecture

The goal of our work is to provide an architecture for multi-user applications that can be scaled to huge numbers of users in a common space, concurrently interacting. In order to do that we need an architecture that allows a great deal of adaptability and resilience, and admits of implementation on a heterogeneous set of resources. Rather than develop highly specific, optimized algorithms, we will create a framework of abstraction that admits of a range of implementations that can be evolved and tuned over time, both within an application, and across applications. This approach is based on the work and writings of David Reed since the mid 70s.

Elements of approach

- Coordinated universal timebase embedded in communications protocol
- Replicated, versioned objects – unifying replicated computation and distribution of results
- Replication strategies – that separate the mechanisms of replication from the behavioral semantics of objects.
- Deadline-based scheduling extended with failure and nesting
- A coordinated “distributed two-phase commit” that is used to control the progression of computations at multiple sites, to provide resilience, deterministic results, and adaptation to available resources. Uses distributed sets.
- Time-synchronized I/O. Input and output to real-time devices are tied to coordinated universal time.

Synchronous vs. asynchronous computing?

The key issue is that we want to be able to provide a continuous experience that coordinates users at multiple locations, interacting in a tightly collaborative way. It isn't unreasonable to expect that all users can see the effect of actions at other sites within 10's of milliseconds.

Consequently, the approach we propose is an architecture that is synchronous to the degree that I/O is synchronized, but at the same time allows for adaptation of computational strategies.

The key idea for I/O coordination is that input and output events (to interactive devices) are synchronized with global universal time, which is coordinated among all sites involved in a computation.

At the same time, objects behave like processes that exist in time, and each object's behavior is implemented by methods that explicitly manage the temporal evolution of the object. In a sense, object internal states are maintained as ordered histories, and operations are performed at "pseudo-time" instants that are properly ordered with respect to I/O operations whose data connect with the objects.

Device I/O is temporally ordered as well. I/O events exist in real time, and provide the coordination between real time and "pseudo-time" that is necessary and sufficient to achieve the proper user interface behavior.

This provides an adaptive approach to real time programming that is not limited to "real time programming".

A Perspective On This Approach

The standard view of a networked virtual environment implementation describes the system as a set of state variables that represent instantaneous system state. Temporal changes are reflected as a sequence of updates to elements of state, and communications distributes the updated state values. This essentially decouples processing from "static" state - that is state that does not change without operation by an external processor that reads and updates it. The model separates processing from storage, and treats consistency as a property of the stored state. Displayed information is then derived from a snapshot of the stored state.

Our view takes Alan Kay's original idea of objects as entities that have behaviors, where messages affect the behavior (state variables are invisible outside the object, and equivalent behavior has meaning independent of how, or even whether, state is represented in any particular way). This allows us to think of self-contained objects that have dynamic behavior even when not driven by external processors. In essence, objects exist in both space and time. Croquet objects interact by exchanging messages. The Croquet view of objects easily incorporates I/O devices, and even real-world objects outside the system, as first class objects in a natural way, whereas modeling objects as abstractions of storage only cannot represent such things as normal objects.

In Croquet, computational time and real time are loosely coupled. The code that executes the dynamic behavior of objects typically can execute a lot faster than the real-time behavior represented, so an object can carry out many seconds worth of behavior per second, if left to itself. The Croquet system's job is to coordinate the execution of objects so that all behaviors that can have a visible effect are completed in time to communicate those effects through the system interfaces.

Since this is the only constraint, objects in the Croquet environment are free to implement a wide variety of strategies for computing their behaviors. As an interesting example, consider an object that receives position reports as messages from a collection of independently flying objects, and displays the positions on a simulated "radar screen". Since all flying objects are simulating their behavior independently, they will compute

their positions as of the real time instant 5:01 pm at quite different computational times. Yet the "radar screen" object can do its job of displaying the state of the flying objects at 5:01 pm by saving all messages it receives, and selecting those position reports that were sent at 5:01 pm as the ones that it displays. This kind of object-specific strategy dramatically reduces the need for lock-step coordination among distributed concurrent activities. Because they maintain some element of past history in the object representations, this kind of approach requires additional storage overhead per object. But the benefit of dramatically better scalability and reduced latency far outweigh the cost of extra storage.

The other key idea in TeaTime is our approach to resilience and fault tolerance. Most large scale distributed virtual environments are quite difficult to handle because at any point in time some elements may become disconnected and other elements may be dynamically added. We recognize this issue in the Croquet object model - each object is responsible for maintaining sufficient information to recover from system disruptions. The key idea in TeaTime is that the state of objects evolve through a distributed *two-phase commit* protocol. Behaviors of all objects that influence each other are first computed, contingent on completion of all dependent object behaviors, and then those behaviors are atomically *committed*. If the behaviors are not completed in time, all contingent calculations are undone by the individual objects.

The principle of giving an object responsibility for its own behavior again allows for a wide variety of strategies for individual objects to implement the proper resilience and recovery. In a networked virtual environment, these strategies can include dynamically adaptive behavior that can cope with heterogeneous hardware, wide variations of delay, and so forth. Applications programmers can tune applications to use new strategies that derive from the unique requirements of their application objects, or use packaged libraries that embed those strategies in abstract object classes that can be specialized for specific implementation

Appendix A: Open Items

Though we have accomplished a great deal in the last year of effort, Croquet is far from complete. This is at least a partial list of the open items that we are focused on for a final release 1.0 sometime in the summer of 2003.

TeaTime and Synchronization

- Final Codewell code
- Working TeaTime model
- Convert existing system to new TeaTime.
- A new user needs to be able to quickly synch his world state up with the rest of the users in a space.
-

Instant messaging interface

Are we using a Jabber client here?

Internet Telephony

Needs a robust interface.

Scripting

- Concept of Green, Yellow, Red access. "Coloring" of messages and presentation tool designed accordingly.
 - Red: Current code, e.g., exposure to all the internals of the system
 - Yellow: "Alice"-functionality, text based, (mostly) high-level interface
 - Green: Tile-based, "impossible" to make (syntactic) mistakes, pre-built viewers

System

- Generalized field based particle system.
- Space based transitions: TCameras and avatars should switch depending upon the space requirements. This would go hand in hand with dynamic behaviors.

Name spaces and Security

- All communication encrypted-all the time – no messages in the clear - ever.
- Session based interlocked key distribution. Concept of Session Owner or Trusted Third Party to provide session keys.
- We need a concept of ownership in general. Looking into selector name spaces for this.
- Clean add/remove model. No evidence that a module existed after it has been removed.
- Dynamic "Through the Portal" class transitions.
- Run time and sources as necessary and as privileged.
- Working with the Environments feature to implement multiple (hierarchical) namespaces. Also pressing forward with extending SystemOrganization and Browser support for managing NameSpaces.

Matrix

- Real speed. Replace B3D routines.
- Required for TeaTime based physics model.
- Need a set of examples to optimize around:
 - The flag and other mesh based transforms (water surfaces).
 - Procedural textures.
 - Bones based mesh transforms.
 - Physic engine – collision detection/tensor math engine

3D Sound

- Full 3D Sound package
- Decide if we can do this in Squeak or if we need to look at alternatives
- Look at OpenAL.

Documentation

- Complete code documentation - every class, every method. Much like the OpenGL reference.
- Developer Documentation - organized programmer guide to the above. Much like the OpenGL Programming Guide.
- Developer Cookbook - example code - cool hacks - etc.
- User Guide - fun with Croquet. For the non-programmer or script programmer – a more complete version of this document..

Component architecture.

- I am not very happy with this. I think I would prefer to have frames as somewhat static objects that can be extended with behaviors dynamically. This is some kind of dynamic, loosely bound mix-in strategy, but I would like some ideas here.

Physics engine.

- Closed form equations and fast collision detection method for complex space curves.

Optimized 3D data format.

- This should be designed to be compact for Internet distribution, but extremely fast to load into the system. We have "standard" compression tools (LZ-family like GZip, Zip etc). However, the largest amount of data is going to come from textures not geometric models (at least for the time being) which means we need JPEG or PNG compression more than anything else. For geometry data we can use straight multi-res streaming representations (trivial stuff).

Additional 3D data importers

- TrueSpace
- Alias
- VRML

Vertex based radiosity.

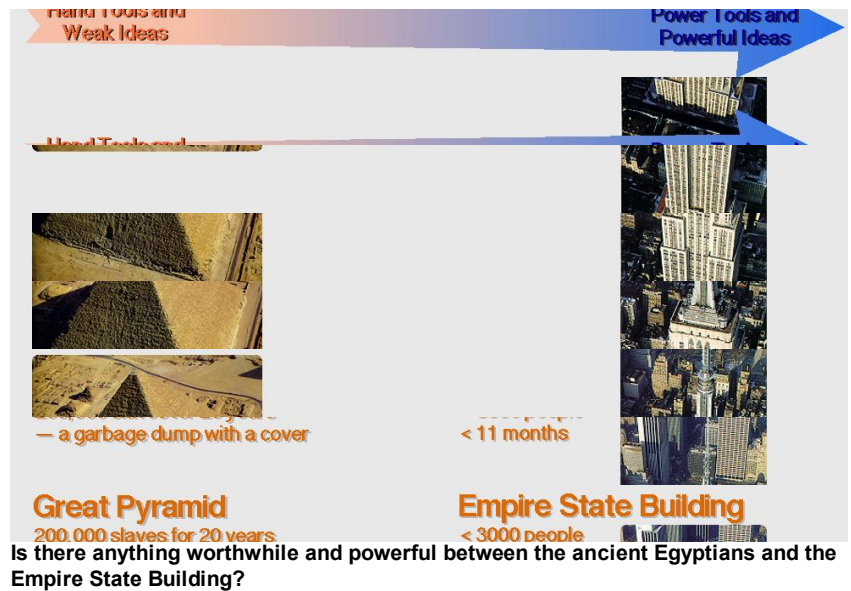
Spatio/Temporal culling.

- We need to develop a method of determining when an object should receive update cycles, especially given that it is shared between users.

Appendix B: Is “Software Engineering” an Oxymoron?

By Alan Kay

Real Software Engineering is still in the future. There is nothing in current SE that is like the construction of the Empire State building in less than a year by less than 3000 people: they used powerful ideas and power tools that we don't yet have in software development. If software does “engineering” at all, it is too often at the same level as the ancient Egyptians before the invention of the arch (literally before the making of arches: architecture), who made large structures with hundreds of thousands of slaves toiling for decades to pile stone upon stone: they used weak ideas and weak tools, pretty much like most software development today.



The real question is whether there exists a practice in between the two—stronger than just piling up messes—that can eventually lead us to real modern engineering processes for software.

One of the ways to characterize the current dilemma is that every project we do, even those with seemingly similar goals has a much larger learning curve than it should. This is partly because we don't yet know what we really need to know about software. But as Butler Lampson has pointed out, this is also partly because Moore's Law gives us a qualitatively different environment with new and larger requirements every few years, so that projects with similar goals are quite different.

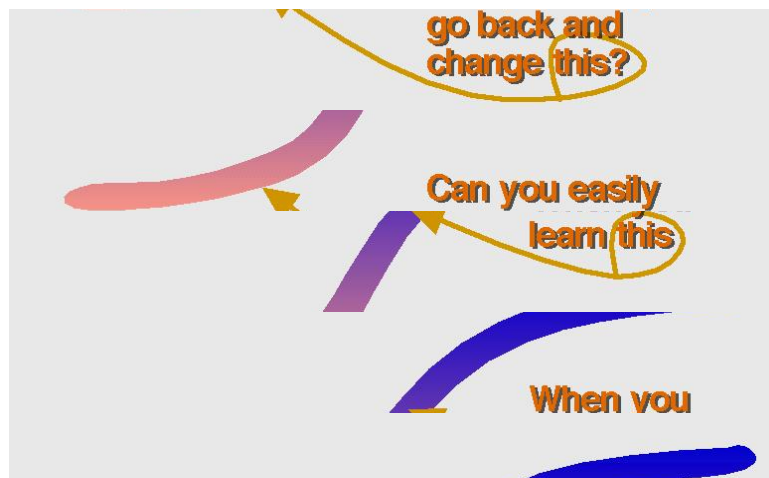
The *Fram oil filter* principle

Very often during a project we'll find out something that we wished we'd known when designing the project. Today, much of the key to more rapid construction and higher success is what we do with this new knowledge. In the "Egyptian" SW systems, we pretty much have to start over if we find a basic new way to deal with the foundation of our system. This is so much work that it is generally not done.

Something quite similar obtains in the ominous fact that 85% of the total cost of a SW system is incurred after it has been successfully installed. The bulk of the cost comes from the changes needed from new requirements, late bugs, and so forth. What seemed expedient and cost saving early on, winds up costing exponentially more when the actual larger life cycle picture is taken into account.

Late binding

Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects. The first system to really do this in an important way was LISP, and many of its great ideas were used in the invention of Squeak's ancestor Smalltalk—the first dynamic completely object-oriented development and operating environment—in the early 70s at Xerox PARC. Squeak goes much further in its approach to late-binding.



Late binding allows ideas learned late in project development to be reformulated into the project with exponentially less effort than traditional early binding systems (C, C++, Java, etc.)

One key idea is to keep the system running while testing and especially while making changes. Even major changes should be incremental and take no more than a fraction of a second to effect. Various forms of “undo” need to be supplied to allow graceful recovery from dangerous changes, etc.

Another key idea is to have a generalized storage allocator and garbage collector that is not only efficient in real-time (so that animations and dynamic media of many kinds can be played while the gc is collecting), but that allows reshaping of objects to be done safely. For example, you have been working on your project for more than a year, and many important things have been built. Can you add several instance variables to a key class that has hundreds of thousands of working instances and have these reshaped on-the-fly without crashing the system? In Squeak this is done all the time, because its storage allocation system was designed for late-bound reshaping, as well as good real-time performance.

Modeling and models of all ideas

A radical key idea is to make the development system have a model of itself so that it can be extended as new ideas occur during development. This means that there are not only classes for workaday tools, such as UI widgets, but there are also classes that represent the metastructure of the system itself. How are instances themselves made? What is a variable really? Can we easily install a very different notion of inheritance? Can we decide that prototypes are going to work better for us than classes, and move to a prototype-based design?

An extension of modeling basic building blocks is to also model and simulate larger facilities. For example, the Squeak Virtual Machine specification is not represented as a paper document (it's not even clear that a 1/2 page paper specification could be made without error just by desk checking). Instead, the Squeak VM specification is represented as a working simulation model within itself that can be run and debugged. New facilities, such as adding FM musical synthesis are not created by writing in C or other lower level languages, but by modeling the new processes in Squeak itself as a module for the VM specification. This is debugged at the high Squeak level (in fact, the VM simulator is still fast enough to run 3 or 4 separate real-time voices). Then a mathematical translator is brought to bear which can make a guaranteed translation from the high level model to low-level code that will run on the two dozen or so platforms Squeak runs on. Further changes are implemented in the VM simulator and become part of the "living spec", then translated as described above, etc.

Fewer features, more *meta*

If we really knew how to write software, then the current (wrong) assumption that we can define all needed features ahead of time in our development system would be sufficient. But since we don't really know how to do it yet, what we need is just the opposite of a feature-laden design-heavy development system: we need one "without features", or perhaps a better way to say this is that we need a system whose features are "mostly meta". Unfortunately, most software today is written in feature-laden languages that don't cover the space, and the needed extensions are not done: partially from ignorance and partially from difficulty.

For example, C++ is actually set up for truly expert programmers, but is generally used by non-experts in the field. E.g. "new" can be overridden and a really great storage allocation system can be written (by a really great programmer). Similarly other features intended as templates for experts can be overridden. It is a source of frustration to Bjarne Stroustrup, the inventor of C++, that most programmers don't take advantage of the template nature of C++, but instead use it as a given. This usually leads to very large, slow, and fragile structures which require an enormous amount of maintenance. This was not what he intended!

Squeak takes a different route by supplying both a deeper meta system and more worked out initial facilities (such as a really good real-time incremental garbage collector and many media classes, etc.). In many cases, this allows very quick development of key systems. However, it is also the case that when deep extensions are needed, many

programmers will simply not take advantage of the many mechanisms for change provided. The lack of metaskills hurts in both languages.

Part of the real problem of today's software is that most programmers have learned to program only a little, at the surface level, and usually in an early bound system. They have learned very little (or nothing) about how to metaprogram — even though metaprogramming is one of the main keys today for getting out of software's current mess.

What are the tradeoffs of learning “calculus”?

Part of the key to learning Squeak is not just to learn the equivalent mechanisms and syntax for workaday programs, but to learn how to take new ideas and reformulate whatever is necessary to empower the system with them. An analogy that works pretty well is to view Squeak as a new kind of calculus. The calculus is not the same as simply a better algebra or geometry, because it actually brings with it new and vastly more powerful ways of thinking about mathematical relationships. Most people will not get powerful in calculus in a month, but many can get a very good handle on some of the powers in a year if they work at it.

One hundred years ago structural engineers did not have to learn the powerful form of calculus called Tensor Calculus, a very beautiful new mathematics that allows distortions, stresses and strains of many different materials (including seemingly empty space) to be analyzed and simulated. But, by 50 years ago, all structural engineers had to get fluent in this new math to be certified as a real engineer. Many of the most powerful and different ideas in Squeak are like those in Tensor Calculus (though easier, we think). They are not part of current practice in spite of their power, but will be not too far in the future.

Most of current practice today was invented in the 60s

It is worth noting the slow pace of assimilation and acceptance in the larger world. C++ was made by doing to C roughly what was done to Algol in 1965 to make Simula. Java is very similar. The mouse and hyper-linking were invented in the early sixties. HTML is a markup language like SCRIBE of the late 60s. XML is a more generalized notation, but just does notationally what LISP did in the 60s. Linux is basically Unix, which dates from 1970, yet is now the “hottest thing” for many programmers. Overlapping window UIs are one of the few ideas from the seventies that has been adopted today. But most of the systems ideas that programmers use today are from the data- and server-centric world of the 60s.

The lag of adoption seems to be about 30 years for the larger world of programming, especially in business. However, the form and practice of Squeak as we know it today was developed in the mid70s and has not yet had its 30-year lag play out. Peer-peer computing was also developed by the ARPA/PARC community around the same period and is just being experimented with in limited ways. Building systems that work the way

the Internet works is not yet part of current practice, etc. (This general observation about the “30 year lag” could be facetious, but it seems amazingly true to history so far.)

Perhaps the most important fact about the new kind of software development as exemplified by Squeak is that a small group of computer scientists have gained quite a bit of experience over the last 25+ years. Thus these are no longer the philosophical speculations of the late sixties, but instead are now a body of knowledge and examples about a wide variety of dynamic computing structures. This makes these new ways much more “real” for those who pride themselves on being practical. For example, a paradigmatic system such as generalized desktop publishing can be examined in both worlds, and some answers can be given about why the late bound one in Squeak is more than 20 times smaller, took even less fractional person-time to create, and yet still runs faster than human nervous systems at highly acceptable speeds. This will start to motivate and eventually require more programmers to learn how to think and make in these “new” ways.

What is it like to learn the new ideas?

How complex does such a flexible facility have to be? It certainly doesn’t have to be complex syntactically. The familiar English word order of Subject Verb Object (often with prepositions and sometimes with an implied subject) is quite sufficient. In object terms, the Subject is the receiver of the message, and the rest of the sentence is the message (we’ll mark the “verb” by darkening it). This allows readable forms such as:

```
3
‘this is some text’
pen up
3+4
3*(4+5)
car forward by 5
{1 2 3 4 5 6 7} collect [n | n odd ]
Repeat (1 to 100 by 2) do [*****]
```

And, just as important, these simple conventions allow us to make up new readable forms as we find needs for them.

This means that a user only has to think of one syntax-semantic relationship.

Receiver message means the meaning is in the receiver

thus objects are thought of and used just like peer-peer servers on a network. (This is not a coincidence and the origin of this idea dates back to the development of the ARPAnet - > Internet in the late sixties).

Thus what we are doing with dynamic object-oriented programming is to design and build a system made of intercommunicating objects. If our object system allows the

interiors of all objects to be themselves made from objects, then we have learned all of the structural knowledge needed.

The interiors can be portrayed as views of the behavioral properties of the object, some of which are more dynamic than others. These behaviors can be grouped into “roles” (sometimes “perspectives” is a good term to use). One role could be “the object as a visible graphical entity”. Another role could be “the object as a carrier/container of other objects. A third role could be “the object as a basic object”. There might be roles that are idiosyncratic to the object’s use in some system. For example, the object could have a special role as a slider widget in a UI.

This is why children and nonprogrammers find this style of programming so particularly easy to learn: it uses a familiar syntax, and has a familiar view of the world, as objects that interact with each other. The objects are made from roles that themselves are collections of relevant behaviors, themselves in terms of objects.

Look at the screen of your computer and realize (perhaps with a shock) that there is really only one kind of object that you are dealing with: one that is represented by a costume consisting of colored regions of space, that can handle user actions, interact with other similar objects, carry similar objects, and may have a few special roles in the “theatrical presentation” of which it is a part. This is all there is from the end user’s perspective. Then why is software so complex? Part of the answer is that normal human nervous systems are attracted to differences, not similarities, and it is similarities that create small descriptions for seemingly large complex systems. Differences often lead to lots of needless special cases that cause increasingly more trouble as time marches on.

“Similarities” are what an algebra does in mathematics: it unifies many seemingly different relationships under a single rubric which represents all. When dynamic object systems were invented in the 60s it was realized that they could have algebraic properties, and it is the algebraic nature of Squeak, combined with the uniform self-contained “objectness”, that accounts for quite a bit of its ability to do a lot with a little. The possibilities of algebra are not at all obvious in the normal old-style early-bound work of work-a-day programming. This is mostly yet to come for most programmers.

Children are also able to deal with parallelism more easily than long strings of sequential logic, so they readily take to writing parallel event-driven systems that deal with interactions dynamically. This leads to small simple programs.

On the other hand, most programmers learn their craft by starting with “algorithms and data structures” courses that emphasize just the opposite: sequential logic and nonobjects. This is a kind of simple “godlike” control of weak passive materials whose methods don’t scale. I think it is very hard for many programmers to later take the “nongod” view of negotiation and strategy from the object’s point of view that real systems designs require.

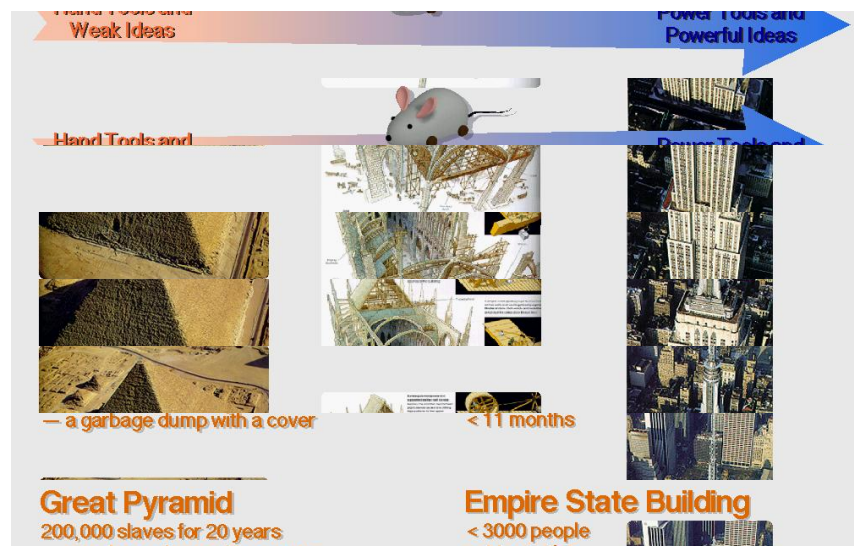
My friend and colleague Dave Reed (whom I think of as the “/” in TCP/IP, because he was instrumental in creating the distinct roles for these two protocols that make the

Internet work) likes to point out that there was a certain odd humbleness in the ARPA, ONR and PARC creators of most of today’s computer technologies. They certainly had the hubris to work on really large problems — such as a worldwide network that could grow by 10 orders of magnitude without having to be stopped or breaking, personal computing for all users, dynamic object oriented system building, a local area net like the Ethernet that is essentially foolproof and dirt cheap, etc. But it was also true that most of these inventors didn’t feel that they were smart enough to solve any of these problems directly with brute force (as “gods”, in other words). Instead they looked for ways to finesse the problems: strategies before tactics, embracing the fact of errors, instead of trying to eliminate them, etc.

Architecture Dominates Materials

In short, you can make something as simple as a clock and fix it when it breaks. But large systems have to be *negotiated with*, both when they are grown, and when they are changed. This is more biological in nature, and large systems act more like ecologies than like simple gear meshing mechanisms.

So the trick with something as simple and powerful as Squeak is to learn how to “think systems” as deeply as possible, and to “think humble” as often as possible. The former is a good perspective for getting started. The latter is a good perspective for dealing with human inadequacies and the need for “metastrategic” processes rather than trying to scale fragile sequential algorithmic paradigms where they were never meant to go, and where they can’t go.



One powerful architectural principle, such as the *vaulted arch* — or *dynamic late bound objects* in software — can provide a boost into a much higher leverage practices for designing and building large complex structures while modern scientific engineering is still being developed

Appendix C: Biographies

DAVID A. SMITH



David began his programming life as a corporate analyst at Thermo Electron Corporation, where he worked to develop an enterprise-wide multi-user multi-dimensional hierarchical spreadsheet program in APL. This system enabled the CEO to get a real-time view of the entire business through its sophisticated updating and reporting capabilities.

In 1982, David went to work for Richard Greenblatt and Lucia Vaina as a programmer for Softrobotics, an affiliate of Lisp Machines, Inc. where he worked to develop an expert system for the diagnosis of brain damage using an Apple][as the front end to a Lisp Machine.

In 1984, David moved back to the Special Projects Laboratory at Thermo Electron to work for Stelianos Pezaris (Sutherland-Pezaris headmount and Pezaris Array Multiplier), where he designed a process control application as well as helped to design a multi-processor distributed controller architecture for a robotic PC plating system. The application was used to design the process that the robotic controller carried out. He also developed a full windows and menus framework for the PC and performed his first experiments in real-time 3D on a PC-XT.

David moved to the Thomas Lord Research Center in 1986 as a Staff Scientist working on intelligent object manipulation using robotic tactile sensors, pneumo-elastic and mechanical hands. He also developed a tele-presence system using stereo-optics and a dataglove controlling a Puma-560 robot equipped with the pneumo-elastic hand. This allowed the user to manipulate small objects from a distance with full eye-hand coordination. It also demonstrated the need for force-feedback to the user for him to accomplish any reasonably complex micro-manipulation task.

David has been focused on interactive 3D and using 3D as a basis for new user environments and entertainment for almost twenty years. He created "The Colony", the very first 3D interactive game and precursor to today's "first person shooters" like Quake... except Colony ran on a Macintosh in 1987. "The Colony" won the "Best Adventure Game of the Year" award from MacWorld Magazine.

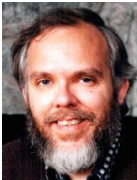
In 1989, David used the technologies developed for the game to create a virtual set and virtual camera system that was used by Jim Cameron for the movie "The Abyss". Based upon this experience, David founded Virtus Corporation in 1990 and developed Virtus Walkthrough, the first real-time 3D design application for personal computers.

Walkthrough won the very first "Breakthrough Product of the Year" from MacUser Magazine.

The Croquet project is the culmination of David's work on 3D component based architectures for the development and deployment of complex peer to peer environments including interactive entertainment. His first experiments in multi-user systems and interactive environments laid the groundwork for much of the architecture and user interface of Croquet.

David co-founded Red Storm Entertainment with Tom Clancy, and Timeline Computer Entertainment with Michael Crichton. He also co-founded Neomar, a wireless enterprise infrastructure company.

DAVID P. REED



David P. Reed enjoys architecting the information space in which people, groups and organizations interact. He is well known as a pioneer in the design and construction of the Internet protocols, distributed data storage, and PC software systems and applications. He is co-inventor of the end-to-end argument, often called the fundamental architectural principle of the Internet. Recently, he discovered Reed's Law, a scaling law for group-forming network architectures. Along with Metcalf's Law, Reed's Law has significant implications for large-scale network business models. His current areas of personal research are focused on densely scalable, mobile, and robust RF network architectures and highly decentralized systems architectures.

Before joining Software Arts, he was a professor of computer science and engineering at the MIT Laboratory for Computer Science, where he helped to shape the early design of LANs and communication protocols. He participated in the design of the protocol suite now used in the Internet and also worked on systems architectures for confederated networks of interconnected personal computers. As a student at MIT, Dr. Reed also was involved in developing commercial implementations of MACLISP and MACSYMA. As a teacher, he helped develop undergraduate and graduate courses in computer and communication systems design, and programming language implementation and design.

Dr. Reed is an independent entrepreneur, advisor and consultant. His consulting practice focuses on businesses that want to capture or create value resulting from disruptive dispersion of network and computing technology into the spaces in which people and companies collaborate and partner.

Prior to moving to Massachusetts, Dr. Reed spent four years at Interval Research Corporation, exploring portable and consumer media technology. For seven years prior to joining Interval, Dr. Reed was vice president and chief scientist for Lotus Development Corporation, where he led the design and implementation of key products, including 1-2-

3, and technical business strategy. Prior to joining Lotus, Dr. Reed was vice president of research and development and chief scientist at Software Arts, the creator of VisiCalc, the first electronic spreadsheet.

Dr. Reed holds a B.S. in electrical engineering and M.S and Ph.D. degrees in computer science and engineering from MIT. Dr. Reed has played a key role on the advisory board of the Vanguard research and advisory program, which was begun in 1991 by CSC Index, and is now a part of the Technology Transfer Institute. He is also a Fellow in the Diamond Technology Partners Diamond Exchange program.

ANDREAS RAAB



Andreas Raab was born on November 24th 1968 in Rostock, Germany. He has been attending the University of Magdeburg (Germany) where he received in 1994 a degree as Diplom-Informatiker (equivalent to Msc in Computer Science) and in 1998 a degree as PhD in Computer Science.

His main background is in Computer Graphics and Interactive Systems. During his PhD research he developed various new techniques for analyzing geometric models (global shape estimation operators), visualizing 3D models (based on real-time non-photorealistic techniques), as well as new interaction metaphors for 3D models (e.g., using zoom techniques for inplace interaction and emphasis in 3D models).

While being at Disney he concentrated on implementing real-time graphics capabilities into Squeak, including support for real-time vector graphics (Flash), real-time 3D graphics, advanced 3D modelling techniques (Teddy), as well as integrating these components into the "eToy" environment (the kids scripting environment).

Besides the work done in the area of Computer Graphics, Dr. Raab also developed an extensive expertise in the area of designing and implementing object-oriented systems. This newly developed knowledge has, for example, helped to redesign the Squeak virtual machine into independent plugins, therefore allowing for an easy extension of low-level operations.

He is fluent on all major computing platforms and programming languages with a few of particular expertise such as C/C++, and Smalltalk. Ever since the original release he has been maintaining the Windows port of Squeak and worked on various other ports (including Windows CE, Sega DreamCast and Sony PS/2).

ALAN C. KAY



Alan Kay, President of Viewpoints Research Institute, Inc., is best known for the ideas of personal computing, the intimate laptop computer and the inventions of the now ubiquitous overlapping-window interface and modern object-oriented programming. His deep interests in children and education were the catalysts for these ideas, and they continue to be a source of inspiration to him.

One of the founders of the Xerox Palo Alto Research Center, (PARC) he led one of the several groups that together developed modern workstations (and the forerunners of the Macintosh), Smalltalk, the overlapping window interface, Desktop Publishing, the Ethernet, Laser printing, and network "client-servers."

Prior to his work at Xerox, Dr. Kay was a member of the University of Utah ARPA research team that developed 3-D graphics. There he earned a doctorate (with distinction) in 1969 for the development of the first graphical object-oriented personal computer. He holds undergraduate degrees in mathematics and molecular biology from the University of Colorado. Kay also participated in the original design of the ARPANet, which later became the Internet.

After Xerox PARC, Kay was Chief Scientist of Atari, a Fellow of Apple Computer for 12 years, and then for 5 years Vice President of Research and Development at The Walt Disney Company. He then founded Viewpoints Research Institute in 2001.

Dr. Kay has received numerous honors, including the ACM Software Systems Award, the ACM Outstanding Educator Award, and the J-D Warnier Prix D'Informatique. He has been elected a Fellow of the American Academy of Arts and Sciences, the National Academy of Engineering, the Royal Society of Arts, and the Computer Museum History Center. He was also a recipient of NEC's C&C Foundation Prize for 2001.

A former professional jazz guitarist, composer, and theatrical designer, he is now an amateur classical pipe organist.

Index

- “mathematically guaranteed” ports, 5
- 2D**, 18, 20, 21, 29, 30
- 32 bits, 9
- 3D, 4, 5, 15, 18, 19, 20, 21, 24, 29, 30, 31, 34, 47, 59, 60, 70, 71, 79, 80, 81
- Algol, 75
- angular velocity**, 16
- arch, 72
- architecture, 7, 8, 59, 66, 70, 72, 79, 80
- ARPA, 75, 78, 82
- ARPAnet, 76
- author, 5
- authoring, 7
- avatars, 40, 69
- behaviors, 5, 55, 57, 64, 67, 68, 69, 70, 77
- billboard, 34
- bookmarks, 31
- broadband connection, 39
- browser, 43, 44, 47, 50, 52, 53, 59
- C, 74, 75, 79, 81, 82
- C++, 74, 81
- calculus**, 75
- camera, 22, 31, 36, 38, 46, 79
- camera button, 22, 31
- Camera snapshot, 38
- click and drag**, 18
- clock, 59, 60, 61, 62, 63, 64, 65, 78
- close, 12, 45, 62
- closed, 6, 23, 24
- Code, 5
- Cole, 8
- collaboration, 5, 6, 7, 39
- collaborative, 6, 7, 66
- color, 9, 47, 57, 63
- communication, 6, 7, 39, 69, 80
- Communication, 5
- component, 5, 59, 80
- computational time, 67
- computer, 5, 6, 9, 24, 40, 76, 77, 78, 80, 81, 82
- computer software architecture, 6
- conference**, 7, 24, 39
- conference phone call, 7
- connect, 12, 24, 67
- Connect*, 12, 39
- connected, 27, 39
- container, 24, 65, 77
- controls, 36
- CPU, 5
- Croquet, 2, 4, 5, 6, 7, 8, 9, 10, 12, 15, 16, 18, 24, 26, 30, 31, 36, 39, 43, 59, 62, 64, 65, 67, 68, 69, 70, 80
- cross hair**, 15, 16
- CTRL- R, 36
- CTRL-D, 32, 38
- CTRL-O, 37
- cursor**, 15, 16, 18, 21, 34, 40
- Deadline-based scheduling, 66
- delivery, 6
- Desk Top Publishing, 7
- desktop, 7, 9, 28, 30, 76
- development, 4, 5, 6, 7, 72, 73, 74, 76, 80, 81, 82
- development environment, 6
- Display, 9
- distributed two-phase commit, 66, 68
- down arrow button, 22
- drag and drop, 10, 11
- Egyptians, 72
- e-mail, 6
- Empire State building, 72
- encapsulation, 5
- environment, 5, 6, 7, 15, 21, 41, 65, 67, 68, 72, 73, 81
- execute, 10, 11, 67
- exit, 12, 13
- Exiting**, 12
- Exploring, 15, 43
- fault tolerance, 68
- favorites, 31
- Ferrin, 8
- FloatArray, 63
- Force tunneling*, 39
- free, 6, 67
- full screen, 10

game, 15, 25, 27, 79
 garbage collector, 73, 74
 GeForce, 9
 global universal time, 66
glow, 18
 go away, 13
 grab, 18, 20, 22
 Graphics hardware, 4
 GUIs, 7
halo, 21, 22
 hand button, 22
 hardware graphics, 9
 hierarchical, 28, 69, 79
 high bandwidth, 7
 high-speed LAN, 39
 Hillis, 8
 HTML, 75
 icon, 10, 11, 23, 29, 32, 34, 44
 inflated, 34
 Ingalls, 7
 instant messaging, 7
 Internet, 4, 39, 69, 70, 76, 78, 80, 82
 IP address, 40
 Java, 5, 75
 Kaehler, 7
 Kay, 1, 8, 67, 72, 82
 KAY, 82
 Keep, 34
 kill button, 22
 King, 5
 Lampson, 72
 LAN, 39
LAN only (disable Internet)”, 39
 landscapes, 24
 laser, 19, 40
 Late binding, 5, 73
 Late bound, 4
 learning curve, 72
 left mouse button, 15
 links, 6, 28, 31
 Linux, 5, 75
 LISP, 4, 73, 75
 Lord, 5, 79
 low-bandwidth, 6
 Macintosh, 5, 9, 15, 79, 82
 Maloney, 7
 media, 5, 7, 73, 74, 80
 meeting place, 5
 messages, 67, 69
 metastrategic, 78
 Middle Ages, 5
 minus button, 23
 mirror, 16, 17, 19, 21, 24, 25, 36
models, 18, 29, 35, 40, 70, 73, 80, 81
 Moore’s law, 4
 Moore’s Law, 72
 mouse, 15, 16, 43, 48, 54, 75
 moving, 15, 16, 18, 26, 60, 64, 80
 multiplatform, 5
 multi-user network, 6
 native object format, 12
navigate, 15
 navigation, 15, 43
Navigator Dock, 29, 32, 35
 net, 6, 78
 networked virtual environment, 67
 nickname, 39, 40
 nVidia, 9
 objects, 10, 18, 21, 24, 28, 29, 34, 35,
 44, 45, 46, 47, 48, 57, 59, 62, 64, 65,
 66, 67, 68, 70, 73, 76, 77, 79
 open, 6, 24, 25, 44, 69
 OpenGL, 62, 63, 65, 70
 operating system, 4, 39
 OS, 4
 Overhead view, 37
 overlapping window, 7, 82
 Overlapping window, 75
 Oxymoron, 72
 paradigm, 6, 18
 PARC, 7, 75, 78, 82
 parent frame, 60
 partition, 27
 partyname, 39
 peers, 39
 peer-to-peer, 39
 penguin, 16, 17
 penguins, 40
perpendicular, 18
 picking, 15, 26
 place, 24, 29, 40, 41, 44, 46
 platform, 5, 6

plus button, 23
 pointer, 19, 40, 54
 portals, 6, 22, 24, 26, 28, 65
 Portals, 24, 27, 28
 programmer, 15, 70, 74, 79
 project, 7, 8, 9, 28, 31, 72, 73, 80
 Properties, 9, 47
 pseudo-time, 67
 Quake, 4, 5, 79
 RAAB, 81
 real time, 15, 67, 68
 real world, 24
 real-time, 4, 66, 67, 73, 74, 79, 81
 REED, 80
 reference, 22, 63, 65, 70
 Rendezvous, 39
 Replication strategies, 66
 resilience, 66, 68
resize the window, 18
 right mouse button, 15
 room, 4, 8, 24, 26
 Rose, 7
 scalable, 66, 80
 SCRIBE, 75
 Scripting, 43, 69
 sharing privileges, 6
 shift key, 16
 Simula, 75
 Smalltalk, 4, 7, 61, 73, 81, 82
 SMITH, 79
Snapshots, 29, 31, 32
 Software Engineering, 72
 space, 15, 16, 24, 26, 27, 40, 43, 44, 47,
 50, 52, 65, 66, 67, 69, 70, 74, 75, 77,
 80
 Spaces, 24, 29
 spatial connection, 24
spin, 20, 55
 spring-loaded, 16
 Squeak, 6, 7, 9, 10, 28, 29, 30, 59, 61,
 65, 70, 73, 74, 75, 76, 77, 78, 81
 Squeak Virtual Machine, 74
startFullScreen, 10, 11
 storage allocator, 73
 Stroustrup, 74
 synchronized, 66
 synchronous, 66
 TClock, 59, 60, 61, 62, 63, 64, 65
 Tea, 4, 11, 39
 Teapot, 10, 11
 TeaTime, 66, 68, 69, 70
 Technology, 5, 81
 Tensor Calculus, 75
 TFrame, 60, 62, 63, 65
 TGroup, 59, 60
 timebase, 66
 Time-synchronized I/O, 66
tools, 29, 34, 43, 70, 72, 74
 TPainter, 34
 True Color, 9
 user, 2, 4, 5, 6, 7, 15, 17, 22, 24, 25, 26,
 28, 29, 30, 31, 32, 34, 36, 37, 40, 41,
 42, 66, 67, 69, 76, 77, 79, 80
 user interface, 4, 15, 67, 80
velocity, 15, 16
 virtual, 5, 24, 26, 27, 28, 30, 68, 79, 81
 virtual conference rooms, 24
 virtual environments, 68
 virtual location, 24
 Wallace, 7
 web browser, 31
 window, 12, 16, 17, 18, 19, 20, 21, 22,
 23, 26, 28, 31, 34, 39, 40, 41, 42, 45,
 47, 48, 59, 82
 Windows, 4, 5, 9, 10, 18, 81
 world, 4, 5, 6, 8, 15, 18, 20, 24, 25, 26,
 27, 29, 30, 31, 33, 34, 38, 41, 43, 44,
 46, 47, 48, 49, 51, 54, 55, 59, 60, 63,
 65, 67, 69, 75, 77
 World Wide Web, 6
 Xerox PARC, 7, 73, 82
 XML, 75