Version

3

SYGEM SOFTWARE

Java Solutions for a Java World

Jazz3D Manual

SYGEM SOFTWARE

Jazz3D Manual

Table of Contents

Chapter 1

Introduction

Why create another 3D API?

ICON KEY
Valuable information
Important Code
Exercises

Our reasons for creating a 3D API (Application-Programmer Interface) for Java were many-fold. Jazz3D started life as a tiny program to draw triangles to the screen. And that's all it did. Soon, more features were added, like shading and the ability to load 3D models. It quickly became apparent that we could extend this further, creating a full blown 3D API.

This lead us to create version 1 of Jazz3D. This was an excellent learning exercise, but we soon realised that there were many, many improvements which could be made. Version 2 soon followed, representing a huge improvement over version 1 - but it wasn't enough. So, here is version 3 - the latest and greatest.

Another thing that spurred us on to create this API was the complexity of many of the other similar products available. For example, Sun's Java3D is a very capable piece of software, but it is overly complex, and this puts people off using it - especially people with limited programming experience. We wanted Jazz3D to be very easy to use, and have designed it with this in mind.

It wasn't just the complexity of other products that made us create Jazz3D, however. Each available API had it's own set of unique features, and we wanted to create an API which encompassed the best of the rest - and with this release, we believe that we am much closer now to that goal.

Aims and Objectives of Jazz3D

The main aim of this product is to make life easy for you, the Java programmer. The API is designed with the non-programmer in mind (non-3D programmers especially), although it does have enough powerful features to keep experienced programmers happy. Interfaces to objects have been kept as concise as possible, whilst retaining access to all the most important attributes of each object.

Importantly, it is still possible to create a Jazz3D applet with just a few choice commands. The more powerful features of the API are easily accessible, and don’t require any in-depth knowledge of 3D systems or techniques.

Jazz3D also has the aim of being fast - a 3D API must be measured by its speed. However, when designing this API, we didn't want to sacrifice too much ease of use for that speed. Since version 1 and 2, many enhancements have been made to the speed of the system, resulting in large speed increases.

The Future

This new version of Jazz3D contains a large number of the improvements listed in the original manual. Some of them haven't quite made it in yet, and there are other improvements which weren't on the original list.

Here is an updated list of the future enhancements we have planned for Jazz3D v4!

- Shadows in real-time
- More object loaders (including DXF)
- Volumetric fogging
- Phong shading renderer
- Mirrors
- Object morphing
- Bump-mapping
- OpenGL support (using GL4Java)

As you can see, there is a lot of work to do before Jazz3D is truly completed.

Obtaining the Full Version

Jazz3D is a full commercial product. This means that you CANNOT distribute the full product - only the demo may be distributed freely. You can, however, still use it without restriction on your web pages. The demo itself contains all of the features of the full product, with a few bits of text obscuring the image. The only way to remove those is to purchase the full package - one license costs from just $70 (US).

To order the licensed copy of Jazz3D, point your browser at https://secure.element5.com/shareit/checkout.html?productid=139212, and follow the simple instructions! Or, you can find a link to this from our home page - http://www.jazz3d.co.uk

Once we have received confirmation of your payment, we will send you a CD containing the full version of Jazz3D 3, which will give you access to all the features shown in this manual, along with loads of new demonstration programs. Also included on the CD are over 20MB of 3D models.

Chapter 2

Jazz3D Essentials

A quick tour around the basic features of 3D programming

Compatibility

Jazz3D is now fully compatible with all versions of the JDK. There is just one set of class files, written using JDK1.02, which have been tested with JDK 1.02, 1.1.8, 1.2.2 & 1.3.

Because there is now just one set of base class files, there is just one simple 'import' statement you need to make in your programs:

```
import com.sygem.jazz3d3.*;
```

All of this means that Jazz3D is now more compatible than ever before. This import statement will give you everything you require to create a simple Jazz3D program - more advanced packages will be introduced as we progress through this manual.

Dimensions

How many dimensions can you think of? 4 seems to be most people's limit, which is understandable, as we live in a world with 4 dimensions - height, width, depth and time. Jazz3D deals primarily with the first 3 dimensions (hence the 3D part of the name). However, all 3 dimensions allow us to define is a position in space - but what about movement? That's where the 4^{th} dimension comes in, and so Jazz3D also features the 4^{th} dimension - without it we could not create displays which move!

Speed

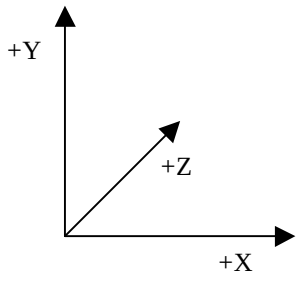
As mentioned in the previous chapter, speed is very important to a 3D engine, and to you the customer as well. We have tried to make this third release of Jazz3D as fast possible - and as time goes by we will try to improve the speed still further. We want this to be the fastest thing there is in Java land.

There are numerous tricks which can be employed to force the best possible speed out of both Jazz3D, and Java as a whole. The most notable being - use a JIT compiler. Both Internet Explorer and Netscape Navigator comes with JIT compilers - as does the standard JDK after version 1.1.2 (for PCs). These can give considerable performance gains. Another trick we came across is avoiding the use of arrays wherever possible - these can be very slow, due to all the bounds-checking Java performs. A large number of array accesses were removed from Jazz3D in the early stages, and sped it up by about 20% instantly!

Within Jazz3D there are also certain techniques which can be employed to squeeze the best performance out of your program. For example, bounding-box checks will not draw an object if the box which contains it is fully off screen. If you use gouraud shading or texture mapping, you will need fewer faces on your object, because of the smoothness of the shading. Finally, beware of using too many light sources - a single directional light source will suffice in many cases.

Basics of 3d engines

Jazz3D uses what is known as a 'left-handed' co-ordinate system. To see what we mean by this, hold up your left hand. Point your thumb upwards - this represents the Y axis. Now point your first finger away from you - this is the Z axis. Your middle finger should then point to the right - the X axis. So, the directions of your digits represent 'positive' - so Z gets bigger the further away from you it gets, and so on.



An object in a 3D environment consists of 2 basic elements - vertex information and face information. The vertex info contains the x, y and z co-ordinates of the point in space. It may also contain data about the vertex normals as well as other data, but the only 3 pieces of information we really need is those x, y and z co-ords.

The face information consists of 3 numbers (for triangles), each of which corresponds to a vertex. This tells the 3D engine which vertices make up the face, and provide a starting point for the rendering pipeline - the heart of the 3D engine. This pipeline starts by translating all the 3D vertex data into 2-dimensional screen co-ordinates. This is where the perspective is applied to the objects.

Once perspective has been applied to the vertices, we can enter the clipping stage. This includes back-face culling (working out which faces cannot be seen, and so should not be drawn), and performing the optional bounding-box check (see Chapter 3 for more details on this). Once this has been done, we then decide if the face can be seen - if all points are to the left or right of the screen, for example, there is no point drawing the face. Following all these checks, we can enter the scan-line rendering phase of the pipe.

This is where most of the cool stuff happens - here is a brief overview of how a scan-line renderer works;

- Setup 2 arrays, one for the left edge, one for the right edge.
- For each face, interpolate between the vertices, filling in the edges
- Draw the face
  - Start at the top - constant y value for each line
  - Draw between the values in the left and right edge tables

And that's it - simple, isn't it! All you ever wanted to know about how 3D engines work. Of course, we could add colour information, shading, textures - whatever we want. But that would be giving away too much...

Chapter 3

The World

What can you do in your new World?

The 'World' object

The very first thing you need to do when creating a Jazz3D applet is create a World. Fortunately, this is made very easy for you (that's half the point of this API!). All you have to do is create a variable of type 'World', then instantiate it.


```
World myWorld = new World();

add(myWorld);
```

Easy, isn't it! All we have done here is create an empty World object. There are no objects, and no lights in your World at the moment - have patience, they will come in time. The other thing to note is that we have added the World to your Application. This is done in the same way as you would add any AWT component - because that's what the World is. It's just an extension of the AWT Canvas - which also makes it easy for you to draw on it yourself (more on that later).

Of course, this can be used in conjunction with any layout manager you choose. Note that if you do not use a layout manager, your world may not stretch to fit the display - be especially wary of this in applets. The best way to ensure your world fits the display is to use the border layout manager (provided as standard with the JDK). Here is an example of how to do this:


```
World myWorld = new World();

setLayout(new BorderLayout());

add("Center", myWorld);
```

Note:

If you are writing Applets using Jazz3D, you will need to use the alternative World constructor, which takes an Applet as a parameter. This is necessary to allow Applet based programs to perform loading operations.

Once the world has been created and all objects etc. have been added to it, there is one final step required to ready the world for display. You need to call the `'prep()'` method before your main program loop begins. This is an essential step - it sets up all the internal variables necessary for displaying the final image.


```
myWorld.prep();
```

It should be noted that a similar method to this needs to be called every time you add a new object to your world with a new renderer object (chapter 4). If you don't, you will find that no objects using the new renderer get drawn to the screen.

```
myWorld.prepNewObjects();
```

World attributes

Bounding Boxes

The default visibility check for objects works on the level of faces - triangles or quadrilaterals. For each object, every face is checked to see if it should be drawn. Whilst this approach makes sense for simple scenes where most objects are visible, it breaks down for more complex scenes, where objects may be off screen at any time.

In complex scenes, it is quite possible that at any point in time, more objects will not be visible than are. In these cases, the default visibility check becomes inefficient - Jazz3D will check the visibility of every face on the object, even if all the faces are off-screen. Once we realised this, we decided Jazz3D needed the ability to check visibility on an object level. That is where the 'bounding box' approach comes in.

For each object, a box that completely surrounds the object is created when the object itself is created. This box is represented by just 8 vertices, which are rotated and translated along with the main object. When the time comes to draw the object, these eight vertices are checked - and if is check is passed, the object is deemed visible. The object is then rendered normally. If all of the vertices are off-screen, then the object cannot be seen, and so is not drawn.

As you can probably tell, for complex scenes with complex objects, this could save a large amount of time. If an object with 1,000 faces is off-screen, rather than check all 1,000 faces, only eight vertices will be checked! This allows more complex scenes to be handled efficiently.

To activate 'bounding box' mode, simply use the following commands;

```
myWorld.setBoundingBoxes(true);

myWorld.setBoundingBoxes(false);
```

Mouse Tracking

This facility gives Jazz3D the very useful ability to become interactive. Basically, Jazz3D will keep track of the positions of all objects on screen. Using a simple API call, you can find out which object is visible at any screen co-ordinate. In a slight change from version 2, mouse-tracking is now permanently enabled.

Mouse events in Java are event driven, so you will need to add code into your event handler for mouseUp events. To find out which object is at screen point (x, y), use the following API call;

```
myWorld.pickClosest(x,y);
```

The value this returns (an integer) will either be -1 if no object is at that position, or it will correspond to one of the numbers returned when you added your objects to the world. This is actually an advanced form of the collision detection you can find in chapter 13, but for now you can just imagine it as simply tracking the objects on screen. You will need to use the HitPoint class to get any useful values, however.

Backgrounds

By default, the background colour of the applet will be black. Plain black. A bit dull and depressing really. What you really want is colour - right? Good. Jazz3D gives you the ability to have full control over the background of your program.

Note:

Any changes made to the background image will **NOT** be reflected if you use the fogging feature. The background colour will become the same as the fog colour.

The simplest way you can change the background of your program is to set it to a single colour. That is done using the following command.


```
myWorld.setBackgroundColour(255,0,0);
```

That would set the background of your applet to a lovely shade of red. Get the idea? Good.

Of course, colour is all very well, but how about images? That's easy as well, with Jazz3D. With just a simple method call, you can set the background to an image of your choice - it can be tiled in the background, or stretched to fit the whole applet.

This image does not have to be static, however. The World object provides 4 methods to allow you to animate the background image. Here is a list of the methods relating to setting the background image:

Method Name	Parameters	Purpose
`setBackground`	String	Loads in a Texture and stretches (or shrinks) it on to the background.
`setBackground`	Texture	As above, but this method takes an already loaded Texture.
`setTiledBackground`	String	Loads in a Texture and tiles it on to the background.
`setTiledBackground`	Texture	As above, but this method takes an already loaded Texture.
`setPanoramicBackground`	String	Loads in a Texture and displays it on the background. No scaling is performed on the image. If the image is smaller than the World, then Jazz3D will revert to Tiled Background mode.
`setPanoramicBackground`	Texture	As above, but this method takes an already loaded Texture.
`shiftBgLeft`	int	Moves the background image to the left by a specified number of pixels. Note that if you are using panoramic backgrounds, this method will NOT work correctly - it only works on the screen image.

Method Name	Parameters	Purpose
		This applies to all of the `'shiftBg'` methods.
`shiftBgRight`	int	Moves the background image to the right by a specified number of pixels.
`shiftBgUp`	int	Moves the background up by a specified number of pixels.
`shiftBgDown`	int	Moves the background image down by a specified number of pixels.
`shiftPanoramicBackground`	int x, int y	Used to move a panoramic background. The parameters represent an offset from the top-left corner of the image. If a number larger than the image size is used, the values simply wrap round, so you don't need to worry about that…

Producing the final image

Inside of your main program loop (see Appendix A for a sample program), after all of your objects have been rotated, translate etc., you need to tell your world object to redraw it's display. There are two ways of doing this - a simple way, and a slightly more complex way. Why the more complex method? Well, it became necessary to support the VisiMagik image processing system (see Chapter 14 for more details). The simple method is achieved with a single method call;


```
myWorld.redraw();
```

By using the second method of redrawing the screen, you gain access to some of the more powerful features of Jazz3D. For example, you can use VisiMagik, or you can use dynamic textures as a background image, or draw directly to the screen image using standard AWT Graphics commands.

```
myWorld.prepareCanvas();

myWorld.generateImage();
```

```
myWorld.drawImage();

myWorld.finishCanvas();
```

So why the extra commands for VisiMagik? Well, to cut a long story a bit shorter, it avoids the need to draw the image to the screen twice. `'generateImage()'` creates an internal representation of the image, ready for processing. This can then be passed to VisiMagik, then once it is processed we draw it to the screen using `'drawImage()'`.

Inside these methods, Jazz3D will produce the image, calling the relevant renderers, then sleep for a period of 20 milliseconds. This allows other processes running on your computer to execute - Jazz3D tries to be nice to other users. You can lower (or raise) this value if you wish. Lowering it can speed up your program, but it will use more system resources as a result. Here's how to set the delay:


```
myWorld.setDelay(10);
```

The parameter is just an integer specifying the number of milliseconds Jazz3D should sleep for. For the fastest possible execution, set the delay to zero. And for more speed increases, you can set the priority of the Jazz3D thread to maximum.

Resolution

The size of the final image is obviously dependent to a large extent on the size of the world - this in turn is determined by both Applet size, and any other AWT components you add to your Applet. However, the size that each object is drawn at is determined by one major factor - the width of the world component. So, if your Applet is 400 pixels wide (set in the HTML file), it doesn't matter how high the Applet is, anything in the world will appear at the same size. In other words, the aspect ratio is always maintained, so the images always look 'normal'.

Special effects

There are a few ways to improve the quality of the final image from within Jazz3D itself. These often require a large amount of processing power, so are not usually suited to general use - but can be used to generate a high-quality final image once your scene is complete.

The first of our ‘special effects’ is bilinear filtering. This can be used to increase the quality of texture mapped objects, by smoothing the image as it’s being applied to the object. Actually, this only has a real effect if the texture being applied is of a fairly low resolution - high-res textures don’t see much of an improvement.

To activate bilinear filtering, use this command:

```
myWorld.setBilinear(true); // turn it on
myWorld.setBilinear(false); // turn it off again
```

Anti-aliasing can also be used to improve the quality of the final rendered image. Jazz3D features 2 types of anti-aliasing - one is suitable for use in real-time, and the other is best suited for rendering a final image.

The 2 types are defined as constants of the World class:

```
World.AA_BLUR; // simple blur filter
World.AA_SUPERSAMPLING; // high quality mode
```

And to choose between the 2 modes, use the following method call:

```
myWorld.setAntiAliasingMode(World.AA_BLUR);
myWorld.setAntiAliasingMode(World.AA_SUPERSAMPLING);
```

Finally, you will need to be able to turn the anti-aliasing mode on or off:

```
myWorld.setAntiAliasing(true); // turn it on
myWorld.setAntiAliasing(false); // turn it off
```

Cleaning up

Java comes with a built-in "garbage-collector" which has been designed to clean up any objects left in memory which are no longer being used. That's a great idea - it frees us Java programmers from having to deal with things like memory allocation. However, it isn't perfect. In fact, we have found that it tends to lull programmers into a false sense of security regarding memory leaks - they still happen in Java, despite what people may tell you about the garbage-collector.

To ensure that Jazz3D doesn't cause any memory-leaks, we have included a method to clean-up all resources used by the World. This includes all objects, lights, renderers, resource loaders - everything.

```
myWorld.destroy();
```

This could typically be called in the `'stop()'` method of an applet, when the main program thread is killed. Of course, once you have done this, there is no returning to the original state of the program, without reconstructing the entire World from scratch, so be careful!

Registering your World

Hopefully you will be so impressed with Jazz3D version 3 that you will want to purchase the full, registered version (a bargain at only $70). When you do receive your package, on the inside of the CD case there will be a registration code. We also include this on the letter we send with each copy of the CD. This registration code is the key to removing the Jazz3D branding from the display.

To apply the registration code, you must use the following command:


```
myWorld.setRegCode("MY REG CODE HERE");
```

What the registration code does is to remove the Jazz3D branding, provided the program is running on a local machine. If you also want to run your applet across the Internet, you will also require a registration code for each domain you wish the applet to be served from. This means you can write one version of an applet and place it on multiple domains without having to alter anything!

Each domain registration code costs just $10, and are available from our website (http://www.jazz3d.co.uk). To activate a domain registration code, use the following command.

```
myWorld.setDomainRegCode("DOMAIN REG CODE");
```

Resource Loaders

Prior to version 3, all of the resource loading in Jazz3D was performed via the World class in some way. That was why the constructor needed an Applet passed into it - to obtain the document base as a starting point for loading stuff. The trouble with that approach was that it coupled Jazz3D to Applets, making Application development less straightforward. It also meant that Jazz3D was only able to load local files - not files located on other web servers. The good news is that this situation has now changed.

Jazz3D now features a set of resource loaders which are designed to be able to load stuff - images, files etc. - from anywhere in a network. There are 3 things which need generally need loading in Jazz3D - images, fonts and models. Each of these has their own resource loader (in fact, the fonts and models use the same loader, as they are both stored as binary files).

There is a set of methods in the World class for obtaining the default resource loader for each type of resource, and also for setting up alternative loaders.

Images

```
getImageResourceLoader()

setImageResourceLoader(ResourceLoader r)
```

The get method simply returns the currently active resource loader for Images. Once you have the resource loader, you can then call the important method - `loadResource()`. There are 2 forms of the `loadResource` method. One takes a String parameter, which is used to specify a local file. The other form takes a URL as the parameter, and returns an Object. This means that if you wish to use this in your programs, you will need to cast the result of the get operation as an Image before using it. The set method simply sets the Image resource loader to a new type of loader.

Fonts

```
getFontResourceLoader()

setFontResourceLoader(ResourceLoader r)
```

These methods work in much the same way as the Image resource loaders. The only real difference is that the Object returned by the `loadResource` method is not an Image, but an URLConnection. From this, you can obtain the actual content on the URL, as well as information about file size. See the Java Documentation for more details on the URLConnection class.

Models

```
getModelResourceLoader()

setModelResourceLoader(ResourceLoader r)
```

The model resource loaders work in exactly the same way as the font resource loaders - in fact, they use the same URLConnectionResourceLoader class. We will talk more about loading models in a later chapter.

Writing your own...

There may come a time when the default resource loaders are not sufficient for your project's requirements. So, here is a brief explanation of how to create your own resource loader for models, loaded from a JAR file (in fact, that is the situation which led to the creation of this whole loading system!).

The first thing you need is to create a class which inherits from the abstract class `GenericResourceLoader`. This provides some basic functionality.

```
class URLResourceLoader implements ResourceLoader
```

Don't forget that this class will need to be able to see the Jazz3D classes at compile time, so you should import them as normal.

When we create this resource loader, we will be passing in the path to the JAR file in the constructor.

```
private URLClassLoader _loader;

URLResourceLoader( URL jarFile ) {

 _loader = new URLClassLoader(new URL[] {jarFile} );

}
```

This gives us the full access to the JAR file, so we can use it in the loadResource method. As mentioned previously, there needs to be 2 versions of loadResource - one to take a String parameter, and one to take a URL. Here are both versions:

```
public Object loadResource(String n)

throws IOException {

    return loadResource(_loader.getResource(n));

}


public Object loadResource(URL url)

throws IOException {

     return url.getContent();

}
```

In this instance, the one you would use would be the String version - to specify the relative path to a file within the JAR file. This would then format a URL, pass it to the other loadResource method, which would return the content of the URL.

Note:

To be used as a Font or Model resource loader, this would need to be modified to return a URLConnection, rather than the content of the URL in question.

All that is left is to describe how you would use this in practice. Firstly, you must create the resource loader, passing in the URL to the JAR file we want to use. Then we could assign it to be, for example, the Image resource loader. Finally, we should be able to use it in the same way as before.

```
String jarFile = “http://www.sygem.com/test.jar”;

URLResourceLoader loader;

loader = new URLResourceLoader(new URL(jarFile));

myWorld.setImageResourceLoader(loader);

myWorld.loadImage(“images/logo.gif”);
```

Chapter 4

Primitive Objects

The basic building blocks of Jazz3D

Available primitives

Before you can access any of the primitives, it is important to note that in this version of Jazz3D, the primitives have been moved to a separate package. This has 2 main impacts of you, the user. Firstly, you will need an additional import statement at the top of each program which uses these primitives.


```
import com.sygem.jazz3d3.primitive.*;
```

It also has the effect of making the base Jazz3D JAR file much smaller, because we have pulled the primitives into a separate JAR. You only need to distribute the JAR files you actually require, making download times shorter.

This is the list of primitives currently implemented in Jazz3D.

- Line
- Triangle
- Quad
- Cube
- Pyramid
- Cylinder
- Sphere
- Torus
- Checkerboard
- Hemisphere

Create the object

When you create a primitive object, you simply need to instantiate it like any other Java object. All the primitives take at least three parameters, these being the position in your world of the object. For example:


```
Cube3d myCube = new Cube3d(0,0,8);
```

This will create a cube with it's center at position (0,0,8). Some of the other primitives take arguments relating to the number of section the object is made up of (sort of like the resolution of the object). These are described at the end of this chapter.

By default, all objects are created with the same size - 1 unit across in all three dimensions. Also, all objects are created white. This simplifies the interfaces to the primitives and reduces the size of the classes. It also forces the user (that's you) to follow the same sequence for every object - create, resize, re-colour, add to world.

Adding objects to your world

Once the object has been created, you are ready to add the object to the world. This makes the object visible in the world.


```
int objID = myWorld.addObject(newCube);
```

It really is as simple as that. The integer which is returned can be used later on to access the object through the world. Whilst this is not the only way that you can access objects at run-time, it is the most efficient, so it is important that you keep track of this variable if you need it.

Deleting objects from the world

There may come a time in the life of your program when a 3D object is no longer required. Obviously you need some way of deleting the object from memory - otherwise you will run out of memory eventually. Fortunately, Jazz3D provides a number of ways for you to delete an object.

Firstly, you can use the integer returned when you added the object to the world. This is the easiest way to permanently remove an object from your world. For example, to remove the object we just added, we would say:

```
myWorld.deleteObject(objId);
```

It is also possible to delete an object based on it's name (see the next section for how to set this name).

```
myWorld.deleteObject("New Cube 1");
```

Finally, you can delete all the objects in the world in one go. This can be useful for setting up new scenes etc.

```
myWorld.deleteAllObjects();
```

You should be aware that there is no undo feature for these deleting operations - so make sure you really want to delete an object, not just set it to invisible.

Object Attributes

Before you add your object to the world, you may decide to override the default attributes for the object. Currently there are 6 major attributes you can set - these methods work in the same way for every Jazz3D object, maintaining a consistent interface throughout the API.

`Name`
Each object is given a unique numerical identifier when you add it to the world, which can then be used to identify the object at a later date. That's fine, but you can also now specify a String name for each object, and obtain a reference based on that String. The name is set using the `'setName()'` method.

```
newCube.setName("New Cube 1");
```

And, if you ever want to find out the name of an object, there is a corresponding 'get' method. Similarly, you can also find out the identifier of a particular object:

```
String name = newCube.getName();

int ID = newCube.getID();
```

Size

Just as in the real world, in a Jazz3D world size matters. Since every object is created initially with a size of 1 units (in all 3 dimensions), you obviously need a way of changing the size. This is achieved by using `'scaleObject()'`.

```
newCube.scaleObject(double x,double y,double z);
```

The three double precision numbers correspond to the desired size in the x, y and z dimensions.

> Note:
>
> If you have bounding box mode turned on (see chapter 3), then the bounding box is automatically re-sized along with the object.

Colour

All primitive objects are created with their colour set to white. Not very exciting, we know, so Jazz3D provides a way for you to change the colour of your objects, using the `'setColour()'` method call. For example.

```
newCube.setColour(255,0,0);
```

This will set the colour of every face in the object to bright red. Very tasteful. So it's easy to set the colour of a whole object - but that doesn't make the objects look very realistic, does it? What about individual parts of an object? Faces, in other words. This is also not a problem with Jazz3D. You can set the colour of a particular face like so:

```
newCube.setFaceColour(facenum,255,0,0);
```

This does depend heavily on the order in which the faces are created. Detail on this is provided for each primitive at the end of this chapter. Note that for model objects and text, the order or each face cannot be guaranteed, or even calculated on the fly. Some experimentation may be required to allow for effective use of this method.

It is also possible to find out the colour of a particular face, using the 'getFaceColour()' method. This will return you an integer - the first 8 bits represent the blue component, the next 8 bits represent the green part, and the last 8 represent the red part. Here is some example code to get these values out:

```
int i = newCube.getFaceColour(1);
int r = (i & 0xFF0000) >> 16;
int g = (i & 0x00FF00) >> 8;
int b = (i & 0x0000FF);
```

Each call to 'setFaceColour()' forces Jazz3D to recalculate the colours at each vertex - these vertex colours are used to enable the smooth Gouraud shading. This can be a costly process if the object is large, and for multiple face colour changes is a little inefficient. So, if you are wanting to make multiple changes of this type, you should use the following method:

```
newCube.setFaceColourQuick(facenum,255,0,0);
```

That method takes the same parameters as the 'slower' version, but it doesn't re-calculate the vertex colours. To force that to happen, you will need to call this method:

```
newCube.applyFaceChanges();
```

Ambient Colour

The ambient colour of an object is the colour it would appear when there is no light shining on it. This is set using the following simple method.

```
newCube.setAmbientColour(int, int, int);
```

Each integer parameter represents a colour value - red, green and blue respectively. If you don't specify the ambient light colour of an object, the default is 30,30,30 - or a dark grey.

This can also be applied to ALL objects in our world, using the following method:

```
myWorld.setAmbientColour(60,0,0);
```

Visibility

When you add an object to the world, it is always visible by default. However, if you don't want your object to be visible just yet, or you want your object to become invisible for a time, this is achieved using the following method;

```
newCube.setVisible(true);

newCube.setVisible(false);
```

Note that this doesn't remove the object from the world - it only tells Jazz3D that it shouldn't be drawn at this point in time. This can be very useful - it means you don't have to delete and then re-create objects at runtime. Of course, all the other attributes of the object remain untouched when using this method.

If you wish to query whether or not an object is visible, you can use this method:

```
boolean visible = newCube.getVisible();
```

Culling

By default, Jazz3D uses a technique called backface-culling to speed up its rendering speed. What this does is to work out if a face on an object is facing away from the camera - if it is, then it shouldn't be drawn. However, sometimes this is not desirable - you may want to see those faces. In this case, you can force the object not to use any culling.

```
newCube.setCulling(false);

newCube.setCulling(true);
```

And if you want to find out if an object is currently using the backface-culling to remove hidden faces:

```
boolean culling = newCube.getCulling();
```

Note:

This has no effect on faces which have been designated double-sided. These faces will always be drawn, no matter which way they are facing.

Object Centers

Direct positioning

Sometimes it just isn't convenient to use the rotate/translate methods to change the position of an object in your world. Wouldn't it be nice if you could just tell the object exactly where it should be, rather than use relative offsets? Well, that's what we thought, so now you can!


```
newCube.setPosition(x,y,z);
```

The three parameters are double-precision variables, and allow you to specify the exact position of your object in WORLD space.

Determine object position

It can also be desirable to find out the exact location of an object. This can be achieved using the following method:

```
newCube.getCenter();
```

This will return a Vertex - you can then use these methods to access the x, y and z components of the Vertex:

```
Vertex v = newCube.getCenter();
double posx = v.getX();
double posy = v.getY();
double posz = v.getZ();
```

This can be useful in determining where a camera or other object should be placed. It could also be used as a starting point for a collision detection ray (see chapter 13 for more details on that subject).

Faces and Vertices

An object in Jazz3D is made up of 2 major components - Faces and Vertices. The vertices specify the location in the world of each point of the object. The faces

represent the 3 or 4 vertices that make up the face. All of these components are now available to you from your Jazz3D programs.

Faces

To get hold of a given face of the object, you need to use the following method:

```
Shape newFace = newCube.getFace(0);
```

The zero here represents face number zero, or the first face created for the object. To find out how many faces are in the object, use this method:

```
int faces = newCube.getNumberOfFaces();
```

This can be very important - if you specify a face number which is out of range, then `'getFace()'` will return `null`, which could cause an exception on your program. It is good practice to check that the face number you requested is less than the value returned by `'getNumberOfFaces()'`.

Once you have the face, there are several important actions you can perform on it. The first thing you can alter is the UV co-ordinates of the face. These are used during texture mapping to determine which part of a texture is displayed on a given face. By default, the UV coordinates of a 4 sided shape are as follows:

Values like this mean that the entire image will be stretched across the face. However, you can specify any values you like - values larger than 1 will make the image tiled across the face, and values smaller than one will only display a portion of the image on the face. Here is the method needed to set the UV co-ordinates:

```
newFace.setUV(0,0,3,0,3,3,0,3);
```

This would set the face to display the image 3 times in both directions - 9 copies of the image in total. The co-ordinates are specified in pairs - U followed by V - in clockwise order, starting from top-left.

A similar operation can be performed for the set of UV co-ordinates used when UV wrapping is being used. More details about the UV wrapping mode can be found in Chapter 6.

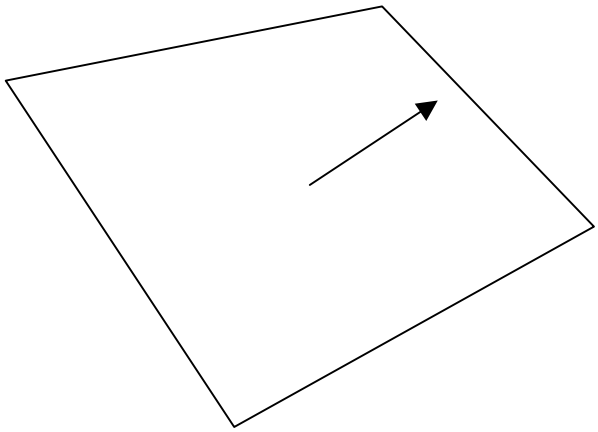
```
newFace.setWrapUV(0,0,3,0,3,3,0,3);
```

You can also query the U and V values of any vertex. Just use these methods:

```
double newU = newFace.getU(1, false);

double newV = newFace.getV(1, false);
```

This will return the U & V values of the first vertex that makes this face. You can also use values of 2, 3 & 4 for the first parameter. Anything other that those will return a value of -1. The '`false`' here indicates that we do not want the wrapped UV co-ordinates - just the standard set of UV values.

The final operation you can perform on an individual face is to alter the 'normal' of the face. The face normal is defined as a vector of length 1, pointing perpendicular to the face. It would look a little like this if you could see it.



The face normals are used to calculate the vertex normals, which are used (amongst other things) for environment mapping (see Chapter 6 for more

on environment mapping). To set the face normal of a face, use this method:

```
newFace.setFaceNormal(new Vertex(0,1,0));
```

It is important that the vertex you pass in to this method be of length 1 - otherwise important calculations may become inaccurate. To retrieve the normal of a face, you can use this method:

```
Vertex normal = newFace.getFaceNormal();
```

Vertices

The vertices of an object can be examined and modified in much the same way that we just dealt with the faces of an object. For example, to get hold of the first Vertex of an object, you would use the following method:

```
Vertex newVertex = newCube.getVertex(0);
```

And to find out how many vertices make up the object:

```
int numVert = newCube.getNumberOfVertices();
```

All vertices of an object are created relative to the center of the object itself. This center is set when you create the object. This can cause a bit of a problem if you want to query the values of the Vertex, because they do not take into account the position of the object. You could get the center point of the object (detailed previously), or you could use the following method:

```
Vertex worldVertex = newCube.getWorldVertex(0);
```

All this really does is add the values of the center of the object onto the values of the Vertex itself. So now you have a Vertex with the actual world co-ordinates of the point - very useful. A Vertex like this could be used as the starting point for a collision detection ray, for example.

So what can you do with a Vertex once you have it? Well, a useful thing would be knowing the location of the Vertex. This can be obtained using the following methods:

```
double posx = worldVertex.getX();
double posy = worldVertex.getY();
double posz = worldVertex.getZ();
```


Note:

Don't forget that if you use `getVertex()` on an object, the Vertex returned has its position relative to the object center, not the World.

It is also possible to set the position of a Vertex, using some similar methods:

```
worldVertex.setX(0.5);
worldVertex.setY(0.25);
worldVertex.setZ(5.26);
```

That is not the only way to set the location of a Vertex, however. It is also possible to set the location to be the same as another Vertex.

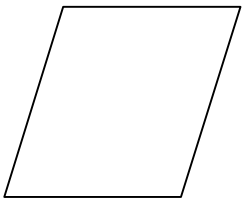
```
worldVertex.set(new Vertex(0.5,0.25,5.26));
```

This doesn’t just assign the new Vertex to the old - that could cause problems. Instead it performs a deep copy of all the relevant data, making sure that although they are different objects in memory, they represent the same location in Jazz3D space.

Lastly, you can also set all 3 co-ordinates directly:

```
worldVertex.set(0.5,0.25,5.26);
```

A word of warning regarding the modification of vertex positioning - it can really mess things up! For example, if a face uses 4 vertices, it expects these vertices to all be in the same plane. If they are not, then equations like backface-culling and even just the rendering itself can go wrong. The same will happen if you cross vertices i.e. make the shape non-convex. Here is an example of what we mean:



Simple convex polygon - This represents no problem to Jazz3D.

Non-convex polygon - this cannot be drawn using a Quad, only using 2 Triangles. This situation could easily arise when modifying vertex positions - so be careful!

In much the same way that we can set the UV co-ordinates for texture mapping for each face, you can also choose to set the UV mapping of a Vertex basis. It's very easy to set the UV co-ordinates of a particular Vertex:

```
myVertex.setUV(0,0); // For example…
```

Note:

If you wish to use the Vertex method of UV mapping, you will need to enable it using the following World method call:

```
myWorld.setUseVertexUVCoords(true);
```

Finally for vertices, you also have access to the Vertex normals. These normals are actually calculated from the face normals - the average direction of the normals of each face consisting of that Vertex (complex, we know, but it works…). These normals are used in the environment mapping process to calculate in which direction the Vertex is looking.

The methods are very similar to those used for faces:

```
Vertex normal = newVertex.getVertexNormal();

newVertex.setVertexNormal(new Vertex(0,1,0));
```

And again, it is very important that the length of the Vertex you pass in is 1. Any other value could cause unexpected results.

Additional Commands

It is now actually possible to get hold of every face and vertex of any given object. This can be very useful for iterating through all faces or vertices

performing additional operations that are not included in the base Jazz3D library. To get all the faces, you can use the following command:

```
Shape[] faces = newCube.getFaceArray();
```

And for the vertices:

```
Vertex[] vert = newCube.getVertexArray();
```

If you browse though the Jazz3D API documentation, you will find that there are corresponding set methods for these operations as well.

Specific primitive information

Line3d

```
Line3d(double x1,double y1,double z1,
       double x2,double y2,double z2);
```

The Line3d primitive is just a single straight line between 2 points. The 2 points in question are specified in the constructor - point 1, followed by point 2.

> Note:
>
> The Line3D primitive can only be drawn using the RenderOutline renderer - any attempts to set another type of renderer will just be ignored. Perhaps in the future we will let you draw texture-mapped lines (or at least Gouraud shaded and transparent lines anyway).

Triangle3d

```
Triangle3d(double x, double y, double z);
```

The triangle primitive gives you just that - 3 points in space, joined to make a solid face. This face is always defined as being double-sided. The constructor takes three parameters which specify the initial position of the triangle. You can then use the following methods to set the position of the 3 vertices:

Note:

You can set the position of each vertex using the following methods:

```
setVertex1(double x, double y, double z);
setVertex2(double x, double y, double z);
setVertex3(double x, double y, double z);
```

Note that these co-ordinates are not relative to the center of the original object, but direct world co-ordinates.

Quad3d

```
Quad3d(double x, double y, double z);
```

Very similar to the triangle class, this gives you 4 points in space, joined to make a single face, also double-sided. The same three methods are used to set the vertex points, and they are joined by one more.

Note:

Be careful when specifying the 4th co-ordinate of this shape - they all need to fall on the same plane in 3d space, otherwise important calculations may fail and the shape will be rendered incorrectly. A good idea is to define the shape with a constant z value, then rotate it as required.

```
setVertex4(double x, double y, double z);
```

Cube3d

```
Cube3d(double x, double y, double z);
```

When used with the Gouraud shader, if you change the colour of any of the faces, the colours spread into each other - this is because of the way the Gouraud shader calculates the colour of a Vertex, by taking the average colour of each face that uses that vertex. This can lead to 'interesting' results - with the Gouraud shader, there can be no sharp changes of colour, unless the object is designed so that no vertices are shared.

Pyramid3d

```
Pyramid3d(int sub,double x,double y,double z);
```

The integer parameter `'sub'` specifies the number of sub-sections the pyramid is to be made up of. An easy way to visualise this is to think of it as the number of sides the bottom of the pyramid is to have. For example, a value of 4 would give a square based pyramid.

Cylinder3d

```
Cylinder3d(int sub,double x,double y,double z);
```

As for the pyramid, the `'sub'` parameter specifies the number of sub-sections that make up the cylinder. So, a value of 4 would actually create a cuboid shape. Obviously, the higher the value, the smoother the cylinder looks.

Sphere3d

```
Sphere3d(int h,int w,double x,double y,double z);
```

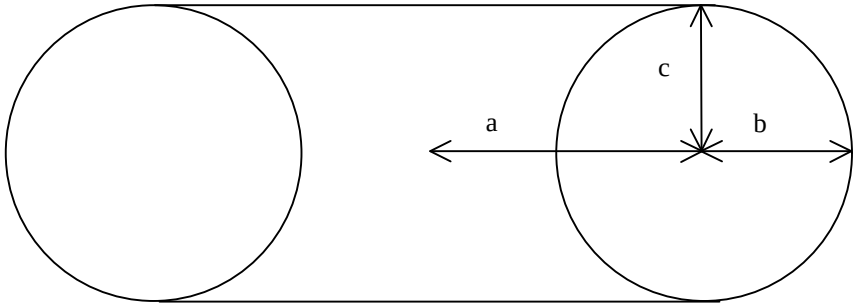
The two parameters `'h'` and `'w'` specify the number of vertical and horizontal sub-sections the sphere is made up of. For a uniform sphere, both numbers should be the same - some interesting objects can be created by altering the values.

Torus3d

```
Torus3d(double a,double b,double c,int w,
        int h,double x,double y,double z);
```

Quite a bit more complex, this one. Like all of the other primitives, the last 3 parameters (x, y & z) specify the position of the center of the object in space. The parameters `'w'` and `'h'` work in the same way as for the Sphere primitive - they specify the number of horizontal and vertical subsections that make up the torus.

The first three parameters are the most important for determining the shape of your torus. This cross-sectional diagram shows how each of the three variables will affect the shape.



So, we can see that `'a'` is the distance from the center of the torus to the center of the ring. `'b'` and `'c'` affect the radius of the actual ring section.

Checkerboard3d

```
Checkerboard3d(int a,int b,double x,
               double y,double z);
```

The two integer parameters of the checkerboard primitive specify the number of squares in the horizontal and vertical directions.

> NOTE:
>
> By default, the colours of the squares are white and red. These can be changed using the `'setColour1(int r, int g,int b)'` for the white squares and `'setColour2(int r, int g,int b)'` for the red squares.

Hemisphere3d

```
Hemisphere3d(int h,int w,double x,
             double y,double z);
```

The constructor for the hemisphere primitive is basically the same as for a normal sphere - the only difference is that this creates a half sphere. `'h'` represents the number of horizontal subsections, and `'w'` is the number of vertical subsections.

Chapter 5

Textures

Add extra realism to your programs using Textures

Loading Textures

The Texture class is basically a wrapper for a standard Java Image, and allows us to extract some useful information for easier retrieval at a later date. This is especially useful for the texture mapping renderers.

Texture loading is achieved through the '`TextureLoader`' object. All the texture object actually stores is the image, in a format more accessible to the internal workings of Jazz3D. To load a texture, just do the following;


```
Texture tx = TextureLoader.loadImage(filename);
```

'`filename`' must be a String representing either the relative path to your image from the applet, or a fully qualified URL to an image. For example, if a file called `'logo1.gif'` was in the same location as the class file for your applet, `'filename'` would just be `"logo1.gif"`. If the same file existed on a website somewhere, `'filename'` might look a bit more like this: `"http://www.jazz3d.co.uk/images/logo1.gif"`.

Supported image formats are (at least) GIF and JPEG. We have heard reports from users of Jazz3D that they have been able to load other image formats - this may be true, but as far as we are aware, only GIF and JPEG are officially supported by Java.

This will load your image into Jazz3D, where you can do one of three things with it: set the background image, load it into one of the renderers which uses images, or pass it to Visimagik for image-processing.

These textures are used by the texture mapping renderers, which use them to map an image onto either each face of an object, or wrapped around the whole object. Information about how to use these renderers is contained in the next chapter.

Animated Backgrounds

Since the background can be set to a texture using just a single method call, Jazz3D also makes it possible to animate these backgrounds. This a very effective way of creating a stunning effect within your Jazz3D programs.

So, to achieve an animated background, you should load in several textures (probably into an array), then inside your main program loop you would simply call the '`myWorld.setBackground(tx)`' method with the next texture in a sequence. All you need to set-up apart from this is a counter variable to access the correct array element each time the loop executes.

You should be aware that if you use `'setBackground()'` at run-time, it may incur a slight performance hit, as each texture needs to be scaled to fit the screen. This scaling is done every time `'setBackground()'` is called, not when the texture is loaded, because when the texture is loaded we don't necessarily know how big the screen is (this is determined by calling `'myWorld.prep()'`).

Of course, you can also use the other methods to set the background texture for tiled textures or even panoramic animated backgrounds!

Texture Sizes

For the first time, you now have the ability to specify the size of any texture within your program. This can be very useful - for example, in Jazz3D version 2, all textures were 256 by 256 pixels, no matter what the size of the original texture was. So, if you tried to load a texture which was 512 by 512 pixels, much of the image information would be lost. When you then came to display this texture on screen, the resolution could appear quite poor.

The decision to scale all textures was made for speed reasons. It makes the texture-mappers considerably faster if the texture size is a power of 2. However, we realised that there was no need to set it at just 256x256. The same code could be used, with no loss of speed, and allow much more flexibility in texture sizes.

As a result, we now present 5 different texture sizes - referenced through fields of the Texture class.


```
Texture.VSMALL (64 pixels)

Texture.SMALL (128 pixels)

Texture.MEDIUM (256 pixels)

Texture.LARGE (512 pixels)

Texture.VLARGE (1024 pixels)
```

So, Jazz3D supports texture sizes from 64x64 pixels to 1024x1024 pixels. This now gives you the ability to load in large textures and not lose important levels of detail in the image.

Not everything is square

But that's not all - you are not limited to square textures any more. This means you can set the size of your texture to one which most closely matches the original image. This is achieved by using a variant of the `'loadImage()'` method introduced at the beginning of this chapter. Actually there are 2 variants: one which takes a single parameter (and will give square textures as a result), and one which takes two parameters (one for the horizontal texture size, one for the vertical texture size).

```
tx = TextureLoader.loadImage(filename, Texture.SMALL);
```

That would create the Texture as being 128x128 pixels.

```
tx = TextureLoader.loadImage(filename,
                             Texture.SMALL,
                             Texture.VSMALL);
```

And that would create the Texture as being 128x64 pixels.

Texture Transparency

Every texture within a Jazz3D program can have a single colour designated as transparent. What that actually means is that at render time, any pixels of that colour will simply not be drawn to the screen (this is only supported for some renderers - see the next chapter for more details).

There are two methods for setting a texture's transparency:

```
tx.setTransparency(16777215);

tx.setTransparency(255,255,255);
```

The first version takes a single integer to represent the transparent colour - this can be worked out using the following simple formula:

transColour = (RedValue * 65536) + (GreenValue * 256) + BlueValue

So, in our example above, both variants of the `setTransparency` method produce the same result - the colour white becomes transparent.

It is also possible to obtain the transparent colour of a texture at a later date. All you need is this simple method:

```
int tCol = tx.getTransparentColour();
```

This will return a single integer variable representing the transparent colour. The RGB is coded in this integer in the same way as explained above for setting it. If you need to extract the individual RGB values, you need some code like this:

```
int r = (tCol & 0xFF0000) >> 16;
int g = (tCol & 0x00FF00) >> 8;
int b = (tCol & 0x0000FF);
```

Texture Mixing

Multi-texturing seems to be the latest thing in computer graphics these days, so you will be pleased to note that Jazz3D now supports a primitive form of multi-texturing. Actually, there are 2 ways to achieve this effect. One is a true multi-texturing (provided by the RenderMultiTextured class, as described in the next chapter). The other way is to mix the textures before they are used in a normal renderer (or as a background image).

The first thing you need are 2 loaded textures. Once you've got that, the rest is fairly easy to pick up! There are 2 basic ways to mix textures prior to rendering them:

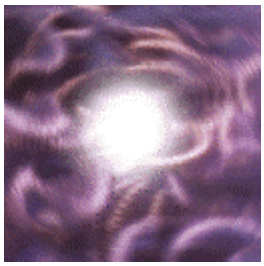
Adding textures

The process of adding one texture to another is very straightforward, and relies on the transparent colour of the texture being added. Here is how to achieve it in code:

```
txBase = myWorld.loadImage(baseImageFile);

txTop = myWorld.loadImage(topImageFile);

txTop.setTransparentColour(255,255,255);

txBase.addTextureLayer(txTop);
```

And here is a real example of the additive process:

Take one base texture…



And a texture with white as the transparent colour…

Add them together, and you get this!

Mixing textures

Mixed textures are slightly different, in that they don't rely on transparency, and take variable amounts of each image to make the final result.

Here is how it would look in your code:

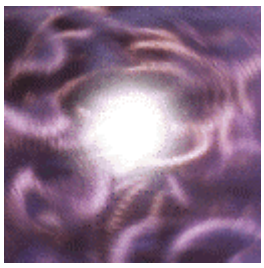
```
txBase = myWorld.loadImage(baseImageFile);

txTop = myWorld.loadImage(topImageFile);

txBase.mixTextureLayer(txTop, 0.5);
```

The important thing to notice here is the number in that last method call. This represents the amount of the base image to mix into the final image (value ranging from 0 to 1).

For example, if you specified a value of 1, then the whole of the base image would be taken, and nothing from the top image. Conversely, if you put a value of zero here, then all of the top image would be used. A value of 0.5 means mix the textures evenly.

Here is a real example, to help explain:

Take one base texture…



And a top texture …

Mix them together (mix value of 0.5), and you get this!

> Note:
>
> When mixing or adding textures, you must ensure that the textures are the same size. If they are not, then the operation will have no effect.

Dynamic Textures

In Jazz3D version 3, you now have complete access to the structure which holds your textures. This opens up all manner of interesting possibilities. For example, you can now use Visimagik (Jazz3D's real-time image processing system) on individual textures. Even better, however, is the ability to create your own textures from scratch. Lets look at a simple example:

First steps

The first, and most important step in creating dynamic textures is to create the texture itself. It is a good idea to create this variable as a global, so you can access it later on. This is very easy:

```
Texture myTex = new Texture(Texture.VSMALL);
```

In this case we are creating a square texture, 64x64 pixels. The size here can be quite important - bear in mind that you will be calculating the values of this texture at runtime. For this texture, that means 64x64 pixels, that's 4096 pixels every frame. If you have some complex calculations, that could become quite a slow operations - so be warned!

Storage Arrays

The easiest way to handle these dynamic textures is to calculate the pixel values into a temporary storage array (which should be the same size as the texture), then copy the contents of this temporary array into the texture. This makes it more efficient for getting and setting pixel values.

Continuing our simple example, here is some more code!

```
int[] tempArray = new int[64*64];

for (int i=0; i<4096; i++) {

     tempArray[i] = 255; // blue…

}
```

So, we have created the array - the same size as the texture itself - and initialised every element to the value 255. Why 255? Well, in our RGB world, 255 means no red, no green, and all the blue you can have!

The next step is to assign the temporary array to the real texture, and assign the texture to be the background image:

```
myTex.setTextureArray(tempArray);

myWorld.setBackgroundImage(myTex);
```

Run-time operation

This is where the extra commands for generating the final display come in handy - remember those? Instead of just saying:

```
myWorld.redraw();
```

We can use a few extra methods to break apart the sequence of events. But before that can happen, we need a method which will update our dynamic texture. All we will do this time is fill the texture with random blue coloured pixels, so it's a nice easy method to write!

```
int[] updateTexture() {
    for (int i=0; i<4096; i++) {
      tempArray[i] = (int)(Math.random() * 255);
    }
    return tempArray;
}
```

With that in place, we can now construct the main program loop to handle the dynamic textures:

```
while (true) {
     // Rotate objects and stuff…
     myWorld.prepareCanvas();
     // Update the dynamic texture
     myTex.setTextureArray(updateTexture());
     // Reset the background image with
     // the new texture array
     myWorld.updateBackgroundImage();
     // Finish drawing objects…
     myWorld.generateImage();
     myWorld.drawImage();
     myWorld.finishCanvas();
}
```

And that about covers it for this simple example! Check out the demos provided for more detailed examples, like how to use dynamic textures with the texture mapping renderers.

Exercise:

Try updating this sample program to produce a red display in place of the blue. You could also try getting the display to be less random - maybe a gradient from black to blue.

Chapter 6

Rendering Systems

How to get your objects to appear on screen

Flexibility is the key

As mentioned briefly in chapter 3, Jazz3D features a very flexible rendering system, behind which is the idea that you should be able to define what type of rendering is performed on each object, rather than the entire world. As a result, Jazz3D's renderers are just Java objects, like everything else in the API. They are created in the same way as other objects, and used just like any other Java variable. This flexibility makes it possible to plug in new renderers at any time, independent of any other classes.

Currently, the following renderers are available:

Renderer Type	Available Styles
RenderOutline	Particles, Anit-aliased lines, image rendering
RenderSolid	Flat shaded, gouraud shaded and environment mapping. Transparency also available.
RenderTextured	Simple texture mapping (perspective-correct) - flat, gouraud and no-shade available
RenderTexturedHQ	Added environment-mapping and transparency
RenderMultiTextured	Additional support for 2-levels of multi-texturing
RenderTransparent	True transparent objects - flat and gouraud shaded

Each renderer is used in much the same way, but many of them have specific functions they can perform, and unique ways in which they can operate. An example sequence for setting up the renderers is as follows:


```
RenderSolid rSolid = new RenderSolid();

rSolid.setDrawingMode(Render.GOURAUD);

rSolid.setTransparency(0.5);
```

RenderOutline

The outline renderer offers 3 different rendering modes.

Particle Mode

The particle render mode is the simplest of all the rendering modes. At each point on an object, a single pixel is draw in the current colour. You should note that mouse tracking doesn't work with this render mode. Also, the colour of each dot is not affected by lighting. This makes it the faster of all the render modes. It therefore makes a good choice for preliminary work, before moving on to the higher quality renderers.

The colour of the dot can be set using the `setPenColour()` method. It takes 3 integer parameters, representing the red, green and blue components of the colour. These integers should range between 0 and 255. For example;

```
RenderOutline particle = new RenderOutline();

particle.setDrawingMode(Render.PARTICLE);

particle.setPenColour(255,0,0);
```

This would set the dot colour to a bright red (and it looks a little like this).

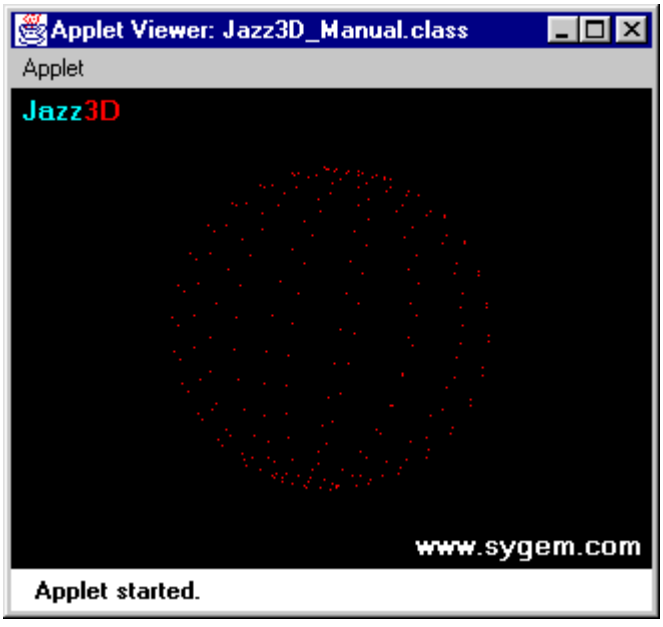


Image Mode

The image renderer is similar in operation to the particle renderer, but it differs in one key area - it is not a single pixel that gets drawn at each vertex, but an image.

After you have created the `RenderOutline` object, you must tell it how big the images it is to draw should be. These can be either 16,32 or 64 pixels in size (those numbers were chosen to speed up the code) - always square images. Here is how to create the `RenderOutline` renderer for each of the 3 possible sizes.

```
RenderOutline imager = new RenderOutline();

imager.setDrawingMode(Render.IMAGE);

imager.setImageSize(RenderOutline.SIXTEEN);

imager.setImageSize(RenderOutline.THIRTYTWO);

imager.setImageSize(RenderOutline.SIXTYFOUR);
```

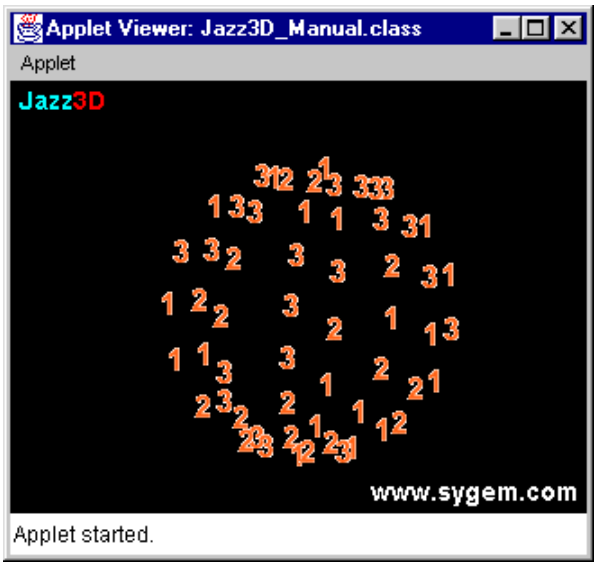
Now you will need to give the renderer the images to draw. This is done through the `'setVertexImage()'` method. It takes a texture object as its only parameter (see the previous chapter for details), and returns an integer value. This can be used to set the image of a particular vertex. For example:

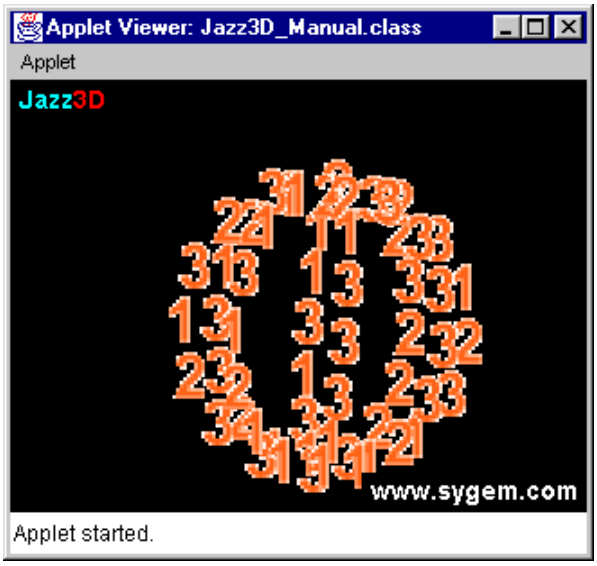
```
int i1 = imager.setVertexImage(tex1);

int i2 = imager.setVertexImage(tex2);

myObject.setVertexTexture(vertexId,i2);
```

This will load 2 textures into the renderer, and tell the vertex numbered `'vertexId'` to draw the second texture. If you don't set the texture to be drawn at a given vertex, it will use the first one you load into the renderer.

Finally, the image renderer supports transparency - a cool effect. Basically, you can designate one colour which will not be drawn from the images you load in. This is set using a method in the Texture class - see chapter 5 for more information.

The 3 pictures that follow show the same object - a simple sphere - rendered with the image renderer at each of the three resolutions (16,32 and 64). You can see that the larger the image, the closer together the images appear. You should be careful that the images don't crowd each other too much.


Applet Viewer: Jazz3D_Manual.class
Applet
Jazz3D
www.sygem.com
Applet started.


Applet Viewer: Jazz3D_Manual.class
Applet
Jazz3D
www.sygem.com
Applet started.


Applet Viewer: Jazz3D_Manual.class
Applet
Jazz3D
www.sygem.com
Applet started.

It can also be seen that the object is not centered on the screen. That is because the image is drawn starting at the position of the vertex. This makes the object shifted to the right and down from it's original location. Again - watch out for this.

Finally, a quick word about performance - the larger the images, the slower the renderer runs. It's as simple as that. The 64x64 renderer is 4 times slower that the 32x32 version, which in turn is 4 times slower than the 16x16 version. Only use the 64x64 renderer if you really need to - and try to reduce the number of vertices in your object.

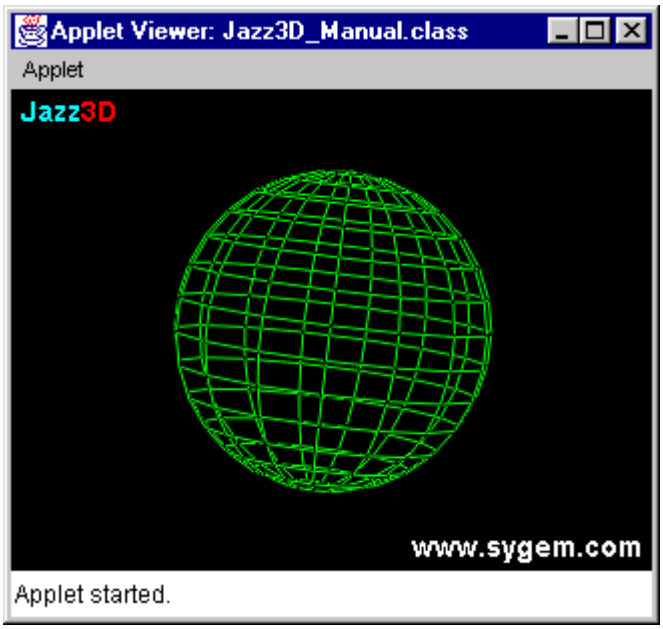
Wireframe Mode

The wireframe mode is also a very simple rendering mode - reminiscent of early arcade games! It is not subject to any lighting, and it only draws the outline of each face. Mouse tracking is not supported in this mode either.

The main specific method call for the wireframe renderer is the setLineColour() method, which allows you to specify the colour used when drawing the lines. It takes 3 integer parameters, representing the red, green and blue components of the colour. These integers should range between 0 and 255. For example;

```
RenderOutline wireframe = new RenderOutline();

wireframe.setDrawingMode(Render.LINE);

wireframe.setLineColour(0,255,0);
```

This would set the line colour to a bright green (very nice!).



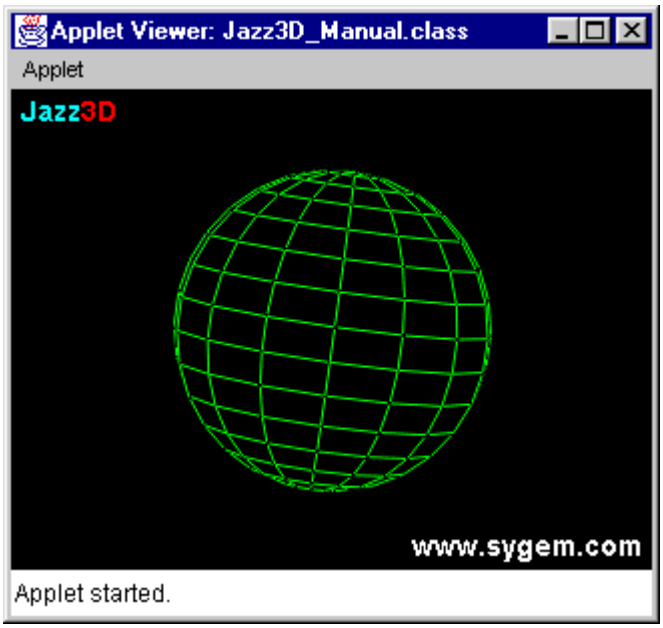
Note:

The default line colour is white - (255,255,255). Also, any objects using the wireframe renderer do not require a lightsource - no shading is performed.

It is sometimes desirable when rendering complex objects using this mode to hide faces which shouldn't be seen - also known as hidden line removal. This can be achieved with the `setHiddenLineRemoval()` method. Like this:

```
wireframe.setHiddenLineRemoval(true);

wireframe.setHiddenLineRemoval(false);
```

Hidden line removal on our example sphere would look a bit like this…

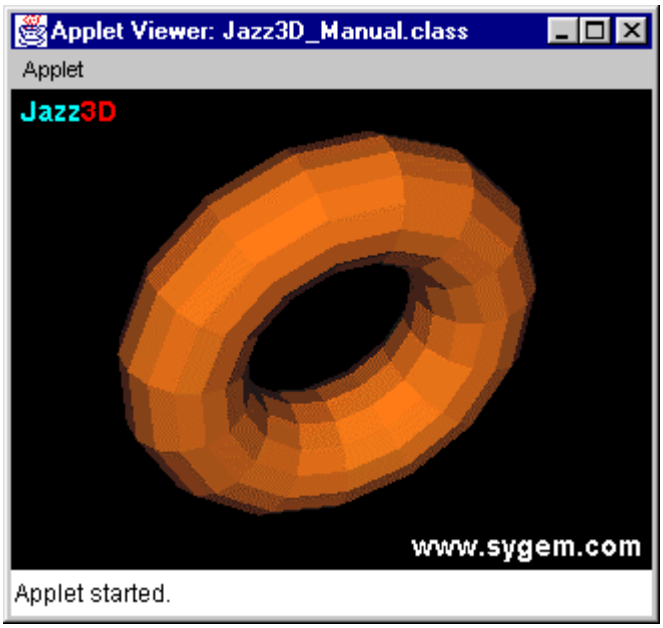


RenderSolid

Flat Shaded Mode

In flat shaded mode, every pixel of a given face is drawn the same colour. This has the unfortunate effect of making the edges of each face very noticeable. However, it is also the fastest of the solid renderers, which does make up for that somewhat. This rendering mode is also sometimes referred to as 'Lambertian' shading.

The lighting calculation for flat shaded mode takes the light intensity at the center point on the face and assumes this value is the same for each pixel on that face. This is then combined with the colour of the face to produce a colour value, again the same over the whole face. There are no specific method calls for the flat shaded renderer.



Gouraud Shaded Mode

Gouraud shading differs considerably from flat shading, not least in the quality of the results that is achieved. Gouraud shading calculates the light intensities at each vertex, then multiplies this by the average vertex colour - the colour of all the faces which use that vertex. It then interpolates these values across each edge, then across each scanline.

RenderTextured

Texture mapping has now become one of the essential elements of any computer game - with the recent advances in video hardware, most of this is achieved using 3D video cards on your PC. Jazz3D's texture mapping is done entirely in software, which means of course that it cannot compete with hardware based texture mapping. It also means that it will not be as fast as that provided by Java 3D - Sun's official release. However, it will run in an applet - which Sun's Java3D will cannot do - and we have tried to make it as fast as possible. Future enhancements to Jazz3D will bring OpenGL™ support, which will allow the use of hardware accelerators.

General use

To assign a texture to the texture-mapper, use the following method;


```
int i = tm.setImage(tex);
```

Note the integer that is returned from `'setImage()'` - this comes in very useful when using multiple or animated textures (described in full later on in this chapter).

For some of the primitives, the option is given to wrap the texture around the whole surface of the object, rather than just on each face. This can give some very pleasing results, and there is no speed penalty! Currently, the following primitives support this texture wrapping:

- Sphere
- Torus
- Hemisphere
- Cylinder
- Pyramid
- Lathe
- Landscapes

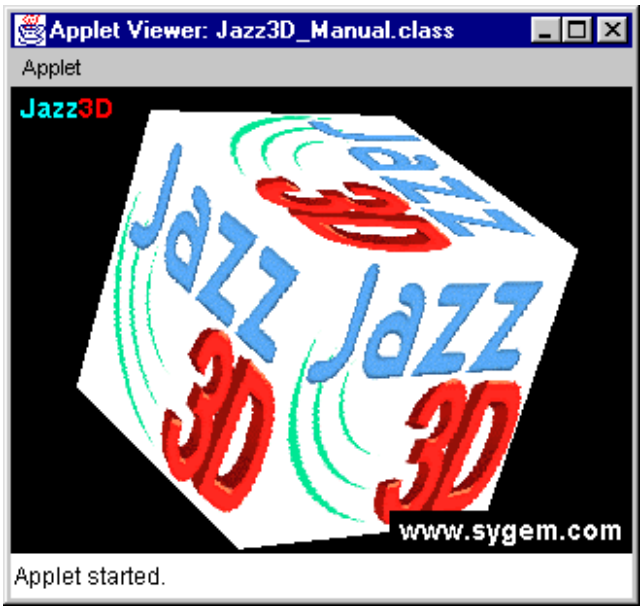
To turn on this feature, just use the following method. Note that if you use this on a primitive which doesn't support the wrapping function, it has no effect (unless you modify the wrapped UV co-ordinates yourself, as detailed in chapter 4).

```
tm.wrapUV(true);
```

The wrapping can be turned off by specifying `'false'` rather than true. Note: this feature can only be activated or deactivated before the object is added to your world - once you have added it, the UV state is fixed.

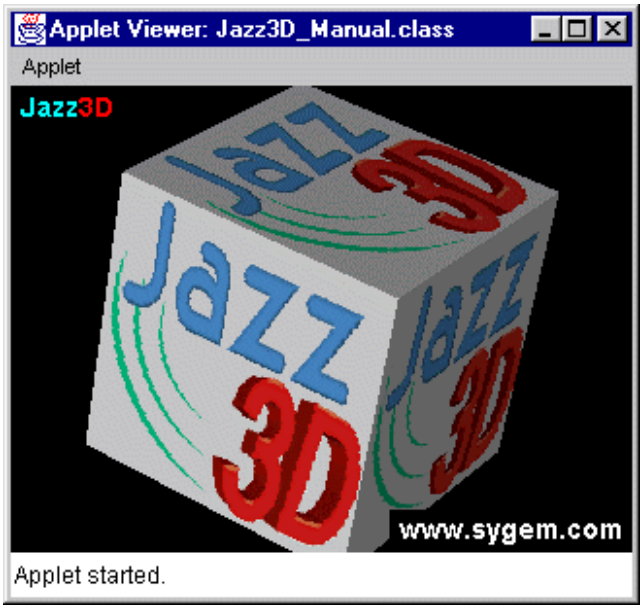
Unshaded Mode

No lighting calculations are performed in this mode, making it by far the fastest of the texture mapping modes. This also means that no light-sources are required to view this object, and as a result the object can look a little flat. However, the speed gains are often worth the sacrifice.



Flat shaded Mode

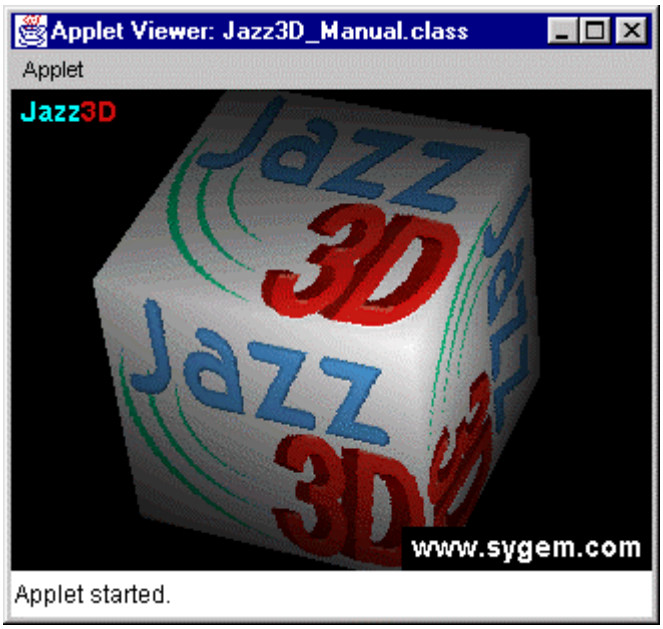
This mode uses the same lighting calculations as the flat shaded mode of the solid renderer, and applies them to a texture.



Gouraud shaded Mode

This texture mapper works in the same way as the previous one - the only difference is in the quality of the image - the approach to shading is that same as for the plain Gouraud shaded renderer.

Here is a picture taken from the same applet, using the Gouraud shaded renderer - I think you'll agree it's a much higher quality image. You can clearly see the smooth changes in colour resulting from this shading type.



Multiple textures

It is now possible to set the texture applied to each face of a Jazz3D object - this is the best way to create really detailed objects. And the best thing is, it's really easy to implement! The first thing you need to do is load your textures. This is described in chapter 3 - then load them into the renderer using the `'setImage()'` method. This will give you an integer which you can use to set the image on a given face. Like this:

```
myObject.setTexture(faceId, texId);
```

Where `'faceId'` corresponds to the face you wish to change. Note that this number starts from zero, not one. Unfortunately, there is often no easy way of working out which face you want to alter, so you may need to resort to trial and error!

It is also possible to set the texture of all the faces in an object with just one method call. This would be useful if you were using the same renderer for multiple objects - you would load in all the textures, then set the texture using the `'setAllTextures()'` method.

```
myObject.setAllTextures(texId);
```

Animated textures

Animated textures can be achieved in real-time - all you need to do is access the renderer of an object at run-time, and use the `'setTexture()'` methods. For example, this loop would set the current texture from an array of 5 texture id's loaded into the relevant renderer.

```
Object3d tObj;
RenderTextured rTex;
while (true) {
     tObj = myWorld.getObject(tObjID);
     rTex = (RenderTextured)tObj.getRenderer();
     rTex.setTexture(texArray[texId]);
     // Could also use this method…
     // tObj.setTexture(texArray[texId]);
     texId++;
     if (texId > 4) texId = 0;
     myWorld.redraw();
}
```

And that's all there is to it! You need to make sure you take note of those texId's!

It is also possible to have animated textures on each face of an object. All you need to do is use a slightly different version of `'setTexture()'`.

```
tObj.setTexture(faceId, texId);
```

RenderTexturedHQ

The High Quality texture mapper (hence the HQ on the end) is a simple extension of the RenderTextured class, offering a few more features.

Environment Mapping

Environment mapping gives us a mechanism by which we can simulate simple reflective surfaces very easily. Using a single image, we can easily calculate which pixel should be drawn at a given location, based on the face normal at that point (a normal is the line perpendicular to a plane).

But you don't really need to know all that - all you need is how to use it, right? Well, the images are set up in a similar way that they are for the texture mappers - using the following method:

```
int envMapID = rTexHQ.setEnvironmentMap(image);
```

Just like the texture mappers, this will return an integer, making it possible to have animated environment maps! How? Well, it's just like using normal animated textures. Instead of passing a texture to the `'setEnvironmentMap'` method, you pass the integer ID of a texture already loaded into the environment map image buffer. For example:

```
rTexHQ.setEnvironmentMap(envMapID);

rTexHQ.setEnvironmentMap(envMapID2);
```

Here is an example of environment mapping:

Note:

Environment Mapping is also available with the RenderSolid renderer! You don't need to have texture maps to use this cool feature, and the methods are exactly the same.

Transparency

There are 2 types of transparency effects available to you when you choose this renderer. The first is based on the transparent value of the texture being rendered (setting this was explained back in chapter 5). If the current pixel being drawn is equal to the transparent value, then it won't be drawn on screen. This effect can be used to draw 3D sprites on screen.

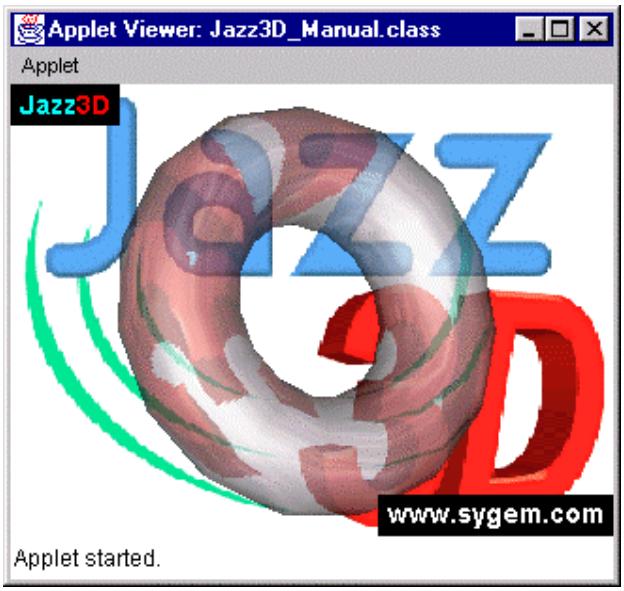
The second style of transparency is much more effective, and gives each face of an object a see-through appearance. It's very easy to set-up as well!

```
rTexHQ.setTransparency(0.5);
```

That would set the renderer to show half the image of the face being rendered, and half from whatever was behind the current pixel. You can set the transparency to be any value between zero and 1 (actually, setting it to 1 would make the object invisible!). To turn off this transparency effect, just set the transparency to zero.

```
rTexHQ.setTransparency(0);
```

Here is an example of the transparency effect in action:



Note:

Once again, this is an effect which is also available when using the RenderSolid renderer. However, for a true transparent effect, you should choose the RenderTransparent renderer - read that section for details on why this is only really a faked transparent effect.

RenderMultiTextured

The multi-texturing available through this renderer achieves much the same results as you would get by pre-mixing the textures (as described in Chapter 5). However, this approach has the advantage of being able to modify both the textures independently of each other, and being able to animate one or both of the textures.

To set up the images for this renderer, you must use the following methods:

```
int texID1 = rMulTex.setTextureBank1(tex1);

int texID2 = rMulTex.setTextureBank2(tex2);
```

To get a better idea of the kind of results you will get from this multi-textured renderer, you should look at the examples given for the static mixing of textures in Chapter 5. The results given by this renderer will be exactly the same, but with the added advantages of being performed in real-time.

Additive Multi-texturing

The additive rendering process involves placed the image from texture bank 2 on top of the image from texture bank 1. Any pixels from the top image which match that image's transparent colour (again, see Chapter 5) will not be drawn, so the bottom image will show through. This is the default mix-mode, but to set it you would use this command:

```
rMulTex.setMode(RenderMultiTextured.ADD);
```

Mixed Multi-texturing

The mixed rendering process mixes the two textures in exactly the same way as the static texture mixing does. It places the texture from image bank 1 on top of the texture from bank 2, but makes the top image transparent, so the bottom image can be seen through the top one. The amount of the bottom image which shows through can be altered (at run-time) using the following method:

```
rMulTex.setMixAmount(0.5);
```

That mix value should be between zero and 1 - the higher the value, the greater the amount of the base image that can be seen. To activate this multi-texturing mode, you should use the following command:

```
rMulTex.setMode(RenderMultiTextured.MIX);
```

Animated Multi-texturing

Just like the other forms of animated textures, this one is very simple to implement. It just involves using slightly different forms of the methods used to set up the original images:

```
rMulTex.setTextureBank1(texID1);

rMulTex.setTextureBank2(texID2);
```

The parameters here are just standard texture Ids, as returned by the `'setTextureBank'` methods. Don't forget that you can only use Ids from the relevant texture bank - you can't go setting texture bank 2 with an ID from texture bank 1, as they may not match up.

> Note:
>
> There is no equivalent method in the Object3D class, unlike for standard animated textures, so you will need to obtain a reference to the renderer at run-time.

RenderTransparent

True Transparency

As mentioned in the previous section about transparency, the effect provided by the other renderers is a bit of a cheat (this is the same kind of transparent effect offered by most other 3D APIs as well). Why is it a fake effect? Well, just imagine a truly translucent cube - you can see all of the faces, can't you? Unfortunately, using the faked transparency this will not always happen - it may happen, but then again, it may not. The problem is that the faces further away from you will not always be drawn - that's why we have this special renderer to enable a true translucency style effect.

When you create your transparent renderer, the default value for the transparency is 0 - this value means that the faces of any object using that renderer are not transparent at all. This can be changed using this method:

```
rTrans.setTransparency(0.5);
```

A value of 0.5 means that the faces become half transparent. If a white face (255,255,255) with 50% transparency was drawn over a red face (255,0,0), the result would be a kind of pinky colour (255,127,127).

The value for transparency can range between 0 (not see-through at all) and 1 (completely see-through). It is worth noting that you can have multiple transparent renderers, all with different values for the transparency. Jazz3D can handle this at no extra cost.

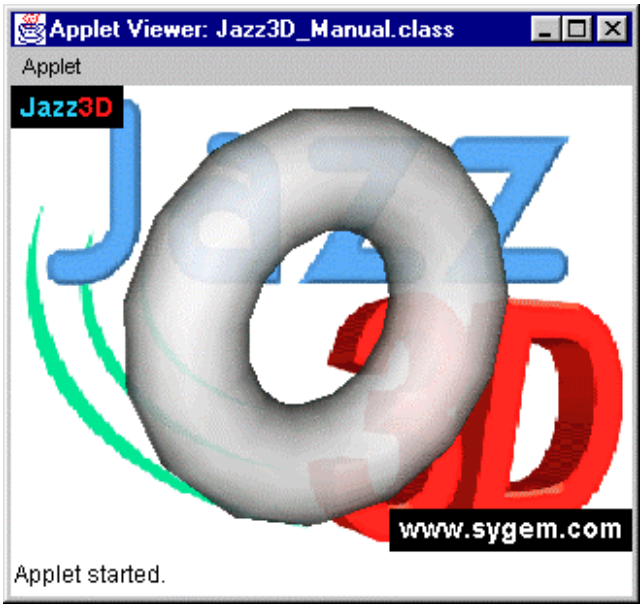
Note:

The transparent renderer can draw objects using both a flat shaded and gouraud shaded modes. These are set in the same way as for all of the other renderers.

Before you render the transparent object for the first time, you need to tell the World of your intentions. This is done using the following method:

```
myWorld.enableTransparentObjects();
```

Here is a screenshot taken using the transparent renderer. There is a slight performance hit - but it can be worth it!



Chapter 7

Pulling it all together

How to put all this together into a simple Jazz3D program

Assigning renderers to objects

So now you know how to create simple objects, and how to create the renderers which will render them to the screen. What you need now is the ability to associate one with the other - and that is done using one simple line of code:


```
myObject.setRenderer(myRenderer);
```

Please bear in mind that you can assign the same renderer to many different objects - this can be very useful for minimising the memory your program consumes. It also makes the program a bit more efficient, and also smaller. However, there are some pitfalls to be aware of when assigning the same renderer to different objects. The key point is that if you change the attributes of the renderer, that change will be reflected across all the objects currently using it. So, if you don't want that to happen, you will need to use separate renderers for each object.

Adding objects to your World

This is probably the most important line of code in any Jazz3D program - without it, there is absolutely no chance that your objects will get drawn. So, here it is!


```
int myObjectID = myWorld.addObject(myObject);
```

Simple really, but very necessary. There are no other forms of this method, so it can't get confused with anything else, and it just works! The important thing to note is the integer that gets returned. This can be used at a later date to get hold of the object, ready for further manipulation at run-time.

Getting hold of objects

There are two methods you can use to allow you to obtain a reference to an object once it has been added to the World. The first returns an object using the integer returned when the object was originally added to the World. The second lets you retrieve an object based on the name of the object - this can be set as described in Chapter 4.


```
Object3d myObj1 = myWorld.getObject(myObjectID);

Object3d myObj2 = myWorld.getObject("Object 1");
```

These methods can also be used to access sub-objects (even though we haven't mentioned hierarchical objects yet) as well as parent objects, by searching through the object tree. However, there is a more efficient way of getting your objects, if you know that they are parent objects (i.e. objects added directly to the World, as shown above). This will only search the top level of the object tree:

```
myObj = myWorld.getParentObject(myObjectID);

myObj = myWorld.getParentObject("Object 1");
```

What happens if the object isn't found?

You can actually decide what should happen if the object you are looking for cannot be found (either the object has been deleted, or the objectID is incorrect). Basically, you can set Jazz3D to return either null (in which case you will need to catch this, because you can't go invoking methods on a null object), or to return a dummy object. If you return the dummy object, it means that any operations you do on it will succeed, but have no actual effect on the system. This is considered a better approach, as it neatly swallows a large number of possible exceptions in your programs - but if your code appears not to be having any effect, this could be the reason!

The method for altering this behaviour is:

```
myWorld.setGetObjectBehaviour(World.RETURN_NULL);
```

And to set it Jazz3D to return the dummy object instead of null, which is actually the default behaviour these days, you should use the following constant:

```
World.RETURN_EMPTY_OBJECT;
```

Let's get them all!

So what happens if you decide that you want to iterate through the whole set of objects that have been added to your world? Easy! Just use this method to get hold of an array of objects:

```
Object3d[] objs = myWorld.getAllParentObjects();
```

However, as the method name might suggest, this only returns the Parent objects - those objects actually added to the World itself. If you want to access sub-objects, you can get hold of them using this method:

```
Object3d[] subs = myObj.getChildObjects();
```

A simple example

To recap what we have seen so far, here is an example of the sort of things you can do in Jazz3D:

Firstly, you need to create the World itself. Because this is an AWT component, like a Button or a Canvas, you can just add it directly to your Applet or Application in the usual way. Following this, you can create your objects and renderers. This simple example program will just display a single sphere in the middle of the display, using a flat-shaded solid renderer. And once they are created, the renderer will get associated with the object, and the object finally added to the World.

All of these action are typically done in the `init()` method of an Applet:

```
public void init() {
   myWorld = new World(this);
   add(myWorld);
   RenderSolid gShader = new RenderSolid();
   Sphere3d sp1 = new Sphere3d(15,15,0,0,5);
   gShader.setDrawingMode(Render.GOURAUD);
   sp1.setRenderer(gShader);
   objId = myWorld.addObject(sp1);
```

But that isn't the only thing we could do in our little demo program. Why don't we load in a texture from disk, and display it as the background to our program.

```
    Texture t = TextureLoader.loadImage("t.gif");
    myWorld.setBackground(t);
}
```

That makes things a little more interesting! Is there anything else we can do to make our rather dull program a bit more exciting? Err… not really, but how about altering some of the object attributes? Because the object has already been added to the World, we will need to get a new handle on it:

```
void modifySphereObject() {
     Object3d myObj = myWorld.getObject(objId);
     myObj.setColour(255,0,0);
     myObj.setAmbientColour(0,0,60);
}
```

This useful little method can be run from anywhere within your Jazz3D program, once the object has been added to the World, and will set the colour of the sphere to red, and the ambient colour to a dark green.

All that is left to do now in this program is the main loop where the rendering occurs - and we will also rotate the object a bit (see the next chapter for more details on this):

```
modifySphereObject();
while (true) {
     Object3d theObj = myWorld.getObject(objId);
     theObj.rotateWorld(0.5,0.23,0.14);
     myWorld.redraw();
}
```

And that really is about it! Of course, this is not a full Jazz3D program yet. None of the thread handling code is present and we have no light source in our World - but we are only half way through this manual, so let's not try to run before we can walk! These things, and many more, will be covered in the remaining chapters.

> Note:
>
> If you want to look at a real example of a program written using Jazz3D, there is one at the end of this manual in Chapter 15. There are also a large number of example programs included with your Jazz3D distribution.

Chapter 8

Moving and Rotating

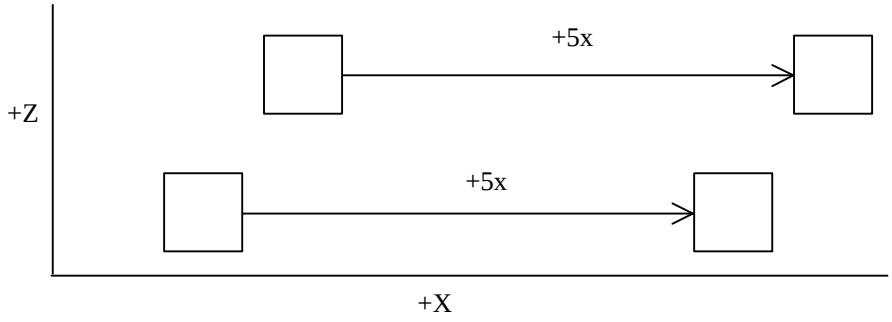
Get those objects in motion!

Moving objects

Once you have your objects created and added successfully to the world, you may well want to move them about. After all, the real world is not a static place, so why should yours be? There are two basic forms of movement in the Jazz3D world - translation and rotation.

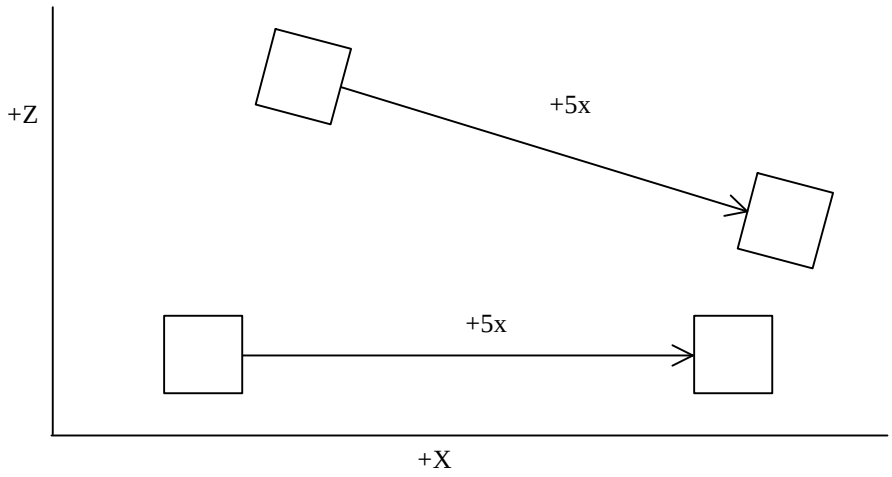
Translation

There are 2 ways to translate the position of an object - you can either do it in World space, or Object space. World space never moves, or changes direction. Translating an object by a given amount will always give the same result in World space. Here is an example:



What we can see is that no matter where an object is (or what direction it is facing) in world space, if you move it the same amount, the same translation will occur. in our example above, a translation of 5 units along the x-axis produces the same movement for both objects.

Object space is a little different, and relates to the direction that an object is facing in. When you rotate an object, it's axis rotates with it. When you then translate in Object space, it follows these newly rotated axis. For example, consider this case:



After the rotation, the axis has changed direction - in other words it is aligned with the object, so the x-axis in object space has been rotated along with the object itself. Translating in world space along the x-axis would still move the object to the right, but a translation along the x-axis in object space will move the object to the right and down a bit. This can take a while to understand (it took us long enough!), so just practice with rotations and translations, and hopefully all will become clear!

Exercise:

Pick up something roughly shaped like a cube (actually, just about anything will do), and rotate it about 45 degrees. Now, translate it in World space, then Object space. Remember that the World space axis never change, but the Object space axis rotate along with the object.

An object can be translated in any of the 3 dimensions - X, Y and Z - through the following simple method calls.

```
myObj.translateWorld(0.5,0,0.67);

myObj.translateLocal(0.5,0,0.67);
```

Of course, the numbers are just examples. As with all of the Object3d methods, you will need to obtain a reference to your object before you use them - the previous chapter had loads of information about that.

Rotation

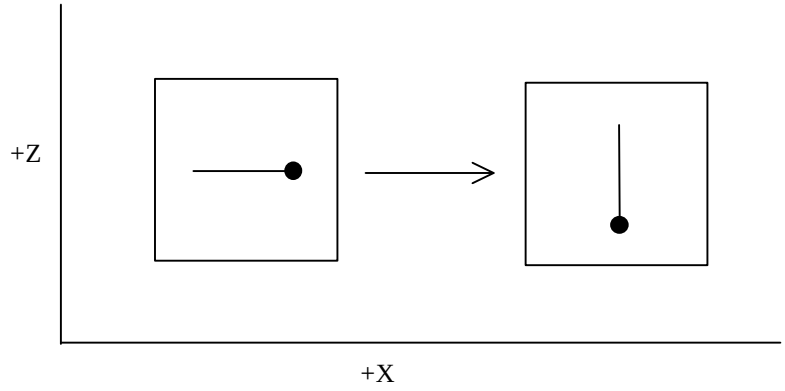
In the same way that an object can be translated about both world and local axis, an object can also be rotated around those same axis. The effect is much the same as for translation as well. There are 2 methods available for rotating around the world axis:


```
obj.rotateWorld(x,y,z);

obj.rotateWorld(x,y,z,0,0,8);
```

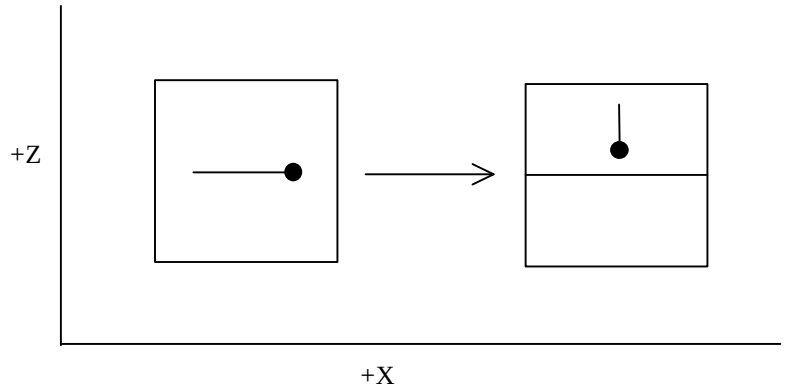
The first method will rotate an object about it's own center point (which is the point you use when you define the object). This method takes 3 integer parameters, representing the number of degrees to rotate in the X, Y and Z axis. This would be very limiting on it's own - fortunate then that we included the ability to rotate an object around any given point, which is what the second method above does.

This is much the same as the previous version of the rotateWorld method - the only difference being the addition of 3 extra numeric parameters (double precision, like in most of these methods), which specify the point you want the object to be rotated around - (0,0,8) in this example.

So, rotating around the world axis has the following effect: Imagine that an object is rotated by 90 degrees about the world y-axis (this is the axis running from the top to the bottom of the screen). Viewed from above, the rotation has the following effect:



Now, what happens if we rotate this object by 45 degrees around the world's x-axis (running from one side of the screen to the other)? The result of this rotation is shown below. Bear this in mind, and note the differences with the same series of rotations performed around the object's axis.



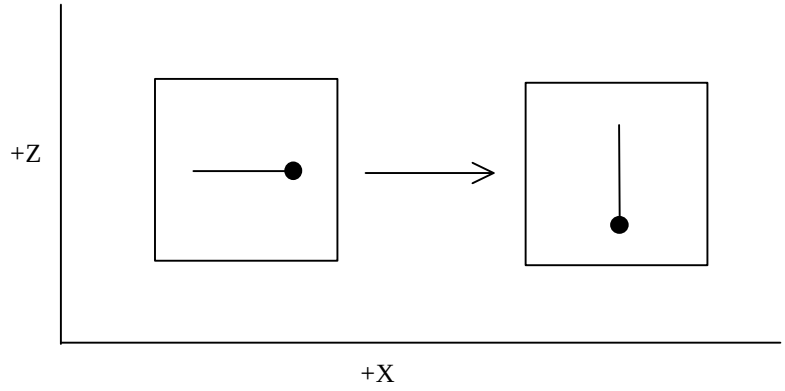
And that sums up the rotation around the World axis - it really is quite simple. All you need to remember is that the axis you are rotating around never move. Which is completely the opposite from rotating around the object's axis.

Rotation about an object's own axis has similar methods to those shown above:


```
obj.rotateLocal(x,y,z);

obj.rotateLocal(x,y,z,0,0,8);
```

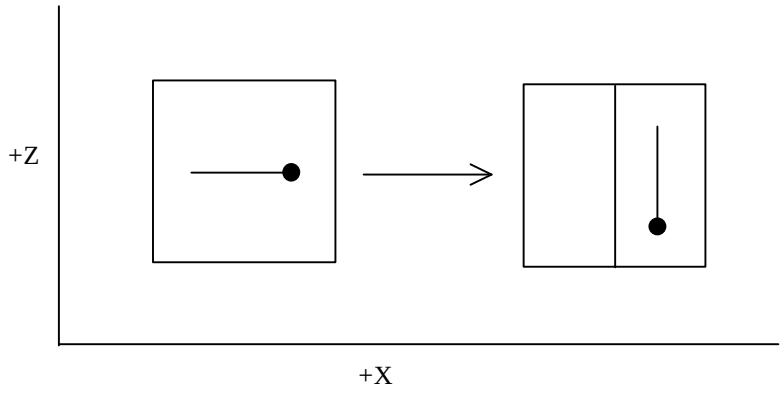
There are methods to rotate around the objects center point, and also around a given point. The only difference is in the effect the rotations have on the object. Bearing in mind the examples shown above for world rotation, lets run through the same steps for local rotation.

Given an object, whose local axis are aligned with the world axis (as they always are when you create an object), we rotate by 90 degrees about the y-axis. The result is shown below:



OK, so that is the same as for World rotation. But the important

difference cannot be seen - the direction of the object's axis - they have rotated along with the object itself. This is the key to the whole object rotating thing. When we now rotate the object by 90 degrees about it's own x-axis, we are rotating about a different axis from before. This leads to the following result:



Because the object's x-axis now points in the same direction as the World's z-axis does, that is where the rotation appears to be around.

This is extremely powerful - it means that no matter where you object is, or what direction it is facing, a rotation about it's own axis will always produce the same result. That fact can be very useful for simulators, for example, because the rotation is predictable is allows you to move an object in a way which more closely models reality.

Exercise:

Try writing a program which will allow you to see the differences between World-space rotation and Object-space rotation.

Chapter 9

Cameras & Lights

Illuminating the world, and viewing it too!

Types of light source

Jazz3D is not limited to just one type of light-source. In fact, it offers a choice of three, each with its own advantages and disadvantages. Here is a brief overview of the three choices.

Directional Light (Light)

A directional light source has no physical location in space - it just points in a given direction. It is best thought of as a really bright light, a long way away, whose intensity is such that it never diminishes in the realms of our world. A bit like a sun, really, except that the light only shines in one direction.

The lighting calculations for this type of light source are the quickest of the three, but the results given are perhaps indicative of this, being not quite as realistic. However, for simple scenes this is the best light source to use.

Point Light (Lightpoint)

A point light source does have a physical location in Jazz3D space. The light from it shines in all directions, and never reduces in intensity. This strikes a useful balance between the speed of a directional light source and the accuracy of the spot light. It is not much slower than the directional light source either, so it's a good all round choice.

Spot Light (Lightspot)

The spotlight is the most 'realistic' of the light sources available. It features both falloff of intensity (the farther away you are, the less intense the light), and definable light cone. These extra features add a considerable amount to the lighting calculations, and investigations have shown that each spotlight can reduce the execution speed of your program by approximately 3%.

Despite this, the results are worth it - especially if you have a fast computer.

Creating a light

All three light classes take just three floating point parameters in their constructor. The basic format is as follows;


```
Light light1 = new Light(0,0,1);
```

For the basic type of light, the three numbers represent the direction vector of the light source. It doesn't matter how large the vector is, only the direction it points in is important.

For the two more advanced light sources, the three parameters represent the location in space of the light source - just as for the primitives described earlier.

Once the light has been created, you may need to set the attributes of the light to something other than the default values. A light is created with a default intensity of 1 (the maximum it can be), and with a default colour of white (255,255,255).

Light properties

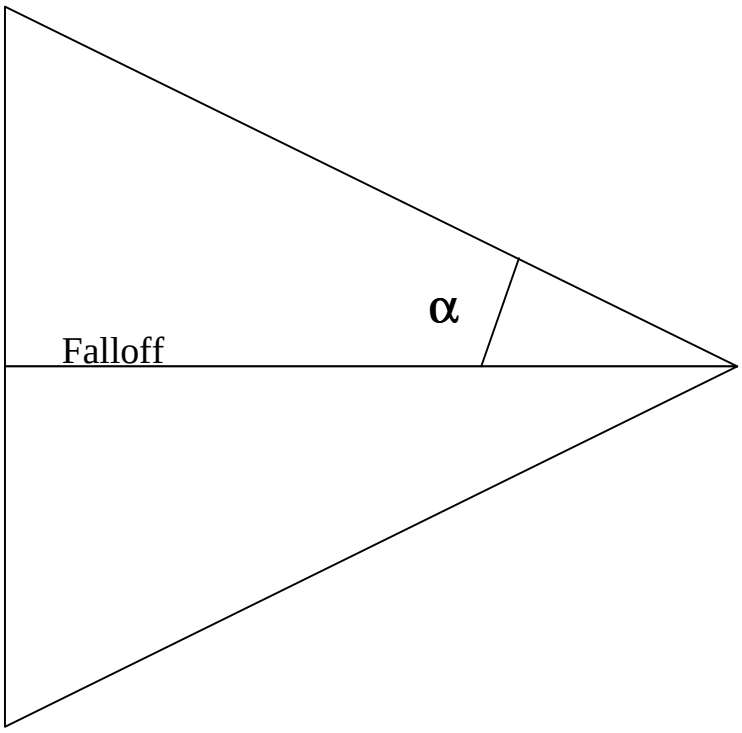
For all light sources, there are two properties you can alter; intensity and colour. The following methods can be used for this purpose.

```
light1.setIntensity(double inten);

light1.setColour(int r, int g, int b);
```

The intensity of a light can range from zero to one. The three values for the colour can range from zero to 255.

Spot Light

The spotlight, being the most complex of the light sources, also features more complex properties than the other two light sources. This diagram shows the light 'cone' of a spotlight, and how you can alter the properties of this cone.



As you can see, there are several properties you can change.

- Angle (α)
- Falloff (distance where intensity reaches zero)
- Direction (not on the diagram, but fairly obvious)

So, to alter the angle of the light cone (which is set to 45° by default), you should use the following method;

```
light1.setAngle(int);
```

The integer parameter specifies the number of degrees the angle (α) is to become.

To alter the falloff of the spotlight cone, use this method;

```
light1.setFalloff(int);
```

As you may expect, the integer parameter represents the distance in Jazz3D units from the origin of the light cone (the brightest point), to the point where the light finally fades away. This transition from bright to no light is linear - at half the length of the falloff, the actual intensity is half the original intensity, and so on.

Adding the Light

Once the light source has been created, it needs to be added to the world, in the same way that objects need to be added. In fact, the whole mechanism works in basically the same way. Here is the method used for adding the light source to your world.


```
lightId = myWorld.addLight(light1);
```

The integer variable lightId can then be used to reference this light source through the world object. To obtain the reference to the light at runtime, you must use the following method:

```
Light myLight = myWorld.getLight(lightId);
```

Light movement

Translation

Translation of light sources only has any meaning for either the point light or spot light - how can you translate a directional light source, which occupies no point in space! Anyway - the method for translating a light source is;

```
myLight.translate(double x, double y, double z);
```

> Note:
>
> The translation of a light source is always done in World space - there is currently no way to translate a light in Object space.

Rotation

Rotation takes on different meanings again, depending on which type of light source you are interested in. Basically, 2 different rotation methods are provided, which are essentially the same as for objects. Here are the interfaces:

```
light1.rotate(int,int,int);

light1.rotate(int,int,int,double,double,double);
```

So, for a directional light, the first rotate method will rotate the direction the light is shining it, around the origin (point 0,0,0). If you remember, this direction is defined as a vector from the origin anyway, so rotating around the origin makes sense. The other rotate method gives you the option of rotating around a given point - this will give slightly different results, which are a little hard to explain. Try to think of the three points in question - the origin, the point you defined the light to shine in, and the point you want to rotate around - that might help.

Because point light sources actually occupy a singe point in space, the rotation is much simpler to explain. You can think of the light as being just like any other object in space - so rotating a point light source using the first rotate method will actually rotate the position around it's center point (as defined when you created the light). However, because the light from a point light sources shines in every direction, rotating the light in this way is meaningless - the light will still shine in every direction, no matter how it is rotated. Rotating the light around a different point will actually change the position of the light, just like for an object, so only this should be used for a point light source.

Spot lights become different again… When you rotate the light using the first rotate method, it's position will not change - in fact, nothing will happen. The second rotate method, rotating around a point, will alter the position of the light source, in the same way as for a point light source. However, the point which the light is pointing at will not change. This must be done using the `pointAt()` method.


```
(Lightspot)myLight.pointAt(0,0,7.5);
```

Limitations

The only limit on the number of light sources you have in your world is memory. There are no hard-coded limits. Do please note, however, that the more light sources you have, the slower your program will run. This is especially true when using spotlights, which have a much more complex lighting algorithm.

Cameras

The camera system of Jazz3D represents a considerable improvement over version 1, which forced you to move all the objects in your world, rather than allowing you total freedom. That is what these cameras represent now - total freedom of movement, with extreme ease.

You always get a default camera object when you create your world. This camera is at position (0,0,0), pointing into the screen (positive z-axis). You can create as many cameras as you like, but your world can only hold one at a time.

To create a camera, just use something like this:


```
Camera3d cam = new Camera3d();
```

This places a new camera at position (0,0,0), again pointing into the screen. But since this is the same at the default camera, I presume you are interested in changing these things… So, to set the position of the camera, you can use either of the following methods:

```
cam.setPosition(Vertex v);

cam.setPosition(double x, double y, double z);
```

So, you can either set the position using a Vertex (such a Vertex can be obtained from any object, or created directly), or using direct world co-ordinates.

Once the camera position has been set, or altered, it is possible to examine the position again, using the following methods:

```
double xpos = cam.getXPosition();

double ypos = cam.getYPosition();

double zpos = cam.getZPosition();
```

Or, you can choose to return a Vertex'- this can be useful for positioning relative to another object.

```
Vertex v = cam.getPosition();
```

Camera rotation

Yes, you can rotate your cameras all over the place! In fact, camera movement is achieved in almost exactly the same way that object movement is done. For example, to rotate the camera, you have the following methods available:


```
cam.rotateWorld(x,y,z);

cam.rotateWorld(x,y,z,0,0,8);

cam.rotateLocal(x,y,z);

cam.rotateLocal(x,y,z,0,0,8);
```

As you can see, the methods are exactly the same as for object rotation. There are methods to rotate the camera around it's center point, and methods to rotate it around any point you choose. The effects of camera rotation are also the same as for object rotation, so you need to be aware of the differences between World rotation and Local rotations.

Note:

Rotating doesn't change the position that the camera is looking at! To do that, use the `'rotatePointAt()'` method.

And for camera translation, the methods again mirror those available for an object:

```
cam.translateWorld(x,y,z);
```

```
cam.translateLocal(x,y,z);
```

As with object translation, the variables x, y & z are double precision numbers, allowing for very precise positioning of your camera. The translation follows the camera's axis - it does not make the camera move towards the point it is looking at.

Point at...

It is also possible to point the camera directly at a given point in space! This means that you could move an object, and have the camera track that object very easily. Here's how:

```
cam.pointAt(myWorld.getObject(objId));
```

So, that version of the pointAt() method takes an Object3d as its parameter. There are 2 other forms of this very useful method - one which accepts a Vertex, and one which takes 3 doubles to specify the point to look at. In the examples below, both lines would have the same effect:

```
cam.pointAt(new Vertex(0,0,8));

cam.pointAt(0,0,8);
```

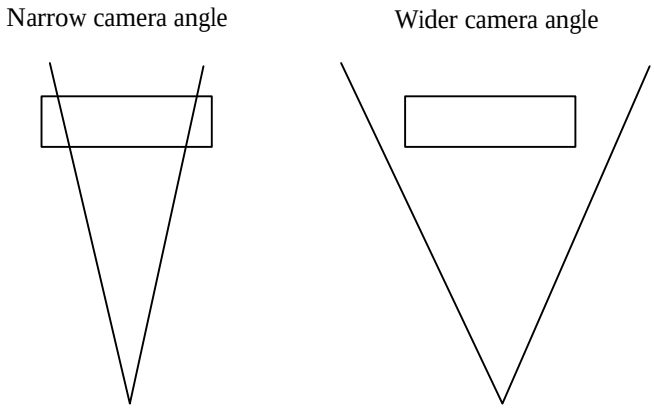
Viewing Angle

The default viewing angle for the camera is currently set to 14 degrees. What that actually means in practice is that the camera can only see things within a certain field of view. Imagine that you had blinkers on - that would alter your perceived field of view - and the same is true for the camera. The value of 14 degrees is mainly for historical reasons.

So, if you would like the field of view to be different, you will need to use this method to change it:

```
cam.setViewingAngle(45);
```

That would set the viewing angle to 45 degrees. Here is a pictorial example to illustrate this concept further:



So, the viewing angle is defined as being the angle between the extremes of your camera's vision - the 2 lines on the diagram above.

Note:

This alteration of the camera viewing angle applies in both the x & y directions - the camera viewing portal always has the same aspect ratio.

World methods

The methods discussed so far were the methods of the camera object itself. However, what you need now is the ability to make Jazz3D use your newly created camera. Fortunately it's a very easy step:


```
myWorld.setCamera(cam);
```

If you need to save the currently active camera before overwriting it with the code above, you can use the following method to get a reference to the current camera:

```
Camera oldCam = myWorld.getCamera();
```

And that's all there is to the subject of cameras. I hope you'll agree that they are surprisingly simple to use, but offer you considerable power. I do recommend that you experiment wherever possible with cameras, as the results can be really impressive.

Chapter 10

Other Objects

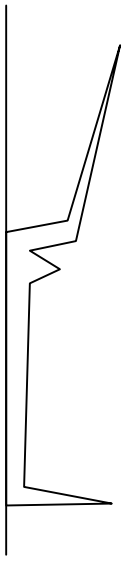
Less primitive primitives!


```
import com.sygem.jazz3d3.primitive.*; // Don’t forget!
```

Lathe Object

To provide a bit more flexibility in your world, Jazz3D includes a `'Lathe3d'` object. This can be used to create some bizarre objects, less uniform than the standard primitives, without needing to load in any external model files.

The idea behind the lathe is that you define key points in two dimensions, and these points are then rotated around an axis to form a solid shape. So, if you had the following shape;



Rotating that around the central axis (the dotted line) would give something approximating a wine glass. Cool, isn't it! So now you need to know how to define these points. It's really very simple. You start with a 2-dimensional array, defined like this:

```
double[][] data = new double[10][2];
```

This will allow us to define 10 key points - note that the second number must be 2, so we can define our points in 2 dimensions. Next, we fill the array with values. The first element of the 2nd part of the array (e.g. `data[i][0]`) stores the x values of the point. The second element of the 2nd part of the array (`data[i][1]`) stores the y value of the point. So, the following data should create something like the wine glass above. It is a good idea to draw your object on paper first, and work out the co-ordinates before you start programming, otherwise it can look a bit confusing on the screen.

```
data[0][0] = 0;
data[0][1] = 0.2;
data[1][0] = 0.1;
data[1][1] = 0.23;
data[2][0] = 0.2;
data[2][1] = 0.4;
data[3][0] = 0.17;
data[3][1] = 0.18;
data[4][0] = 0.05;
data[4][1] = 0.15;
data[5][0] = 0.1;
data[5][1] = 0.10;
data[6][0] = 0.05;
data[6][1] = 0.05;
data[7][0] = 0.05;
data[7][1] = -0.3;
data[8][0] = 0.2;
data[8][1] = -0.35;
data[9][0] = 0;
data[9][1] = -0.35;
```

Once we have our key point data, we can define the Lathe3d object. Here is an example.

```
Lathe3d newObj = new Lathe3d(data, 10, 0, 0, 8);
```

The first parameter in the constructor is the 2-dimensional array we just created. Following this is an integer parameter, defining the number of subsections the lathe will create - the more subsections, the smoother the curve.

The remaining three parameters specify the x, y & z positions of the object in space. Once this has been created, you can just add the object to your world as normal. And that's about it for the Lathe - I hope you can see the potential here!

Fonts

The 3D font feature really came of age in Jazz3D version 2 - it was included as something of a bonus feature in the initial release of Jazz3D. Now, the ability to create 3D text in your world, which can be rotated and moved about just like any other object has become a reality.

The core behind the 3D text support is the font format developed specially for Jazz3D. This contains all vertex and face information for each character the font contains. The font files can be stored in any directory you like - even accessed via the Internet!

To create the font object in your program, you can use the following syntax;


```
Font3d arial = new Font3d("Arial.f3d");

Font3d comic = new Font3d("ComicSans.f3d");
```

This creates the Font3d object and loads in the font file. The string for specifying the font name is case-sensitive, and should represent either the full URL to a font file, or a relative path (relative to your applet or application). It is no longer necessary to pass in your World - that coupling has been removed in Jazz3D version 3.

To check which characters are supported by a font, use the method '`getString()`', as follows;

```
String chars = arial.getString();
```

This will help you reduce errors in your programs, caused by using unsupported characters. Here is a list of all the characters supported by the five fonts included in version 3 of Jazz3D.

20th Century Font:

AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789!$%^&*()-=_+[]{};':@#~,.<>/?\

Arial:
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789!£$%^&*()-=_+[]{};':@#~,.<>/?\|

ComicSans
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789!£$%^&*()-=_+[]{};:@#~,.<>/?\|

Courier:
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789!£$%^&*()-=_+[]{};':@#~,<>/?\|

TimesNewRoman:
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789!£$%^&*()-=_+[]{};:@#~,.<>/?\|

Text objects

OK - so now you have created the font object, you need to get something tangible into your world. This is where the 'Text3d' object comes in. Here's an example of creating a text object.


```
Text3d t = new Text3d("Hi!",font,1,0.1,1,0,0,8);
```

So, the first parameter is the actual text the object is to contain. This is just a String variable - nothing special here. Spaces can be used, along with any of the characters present in the font. If you use a character which is not available, a space will be inserted instead.

The second parameter points to the `Font3d` object you should already have created. Following that is a number which represents the size (in Jazz3D units) of each character in the String. The next parameter after that is a variable which is used to set the distance between each character (again, in Jazz3D units). Following that is a number which specifies the depth of the text - the extrude value.

The final three variables set the position of the `Text3d` object in the world. This represents the center point of the object.

Once the text object has been created, you then just need to add a renderer to it, then you can then add it to your world in the same way as for the primitives.

All of the renderers currently supplied with Jazz3D will work with the Text3d object - even the texture mappers. However, the texture wrapping (UV wrapping) mode will not work - textures will be applied on each face, rather than the whole character.

Fractal Landscapes

This new primitive makes it possible to create wonderful smooth landscapes with ease - and unlike many other landscape generators, this allows you to apply different colours to the landscape, to increase the degree of realism.

The first thing to do with your landscape is create the object. This is done in the same way most of the Jazz3D primitives are created - a little like this:


```
Landscape3d island = new Landscape3d(0, 0, 0);
```

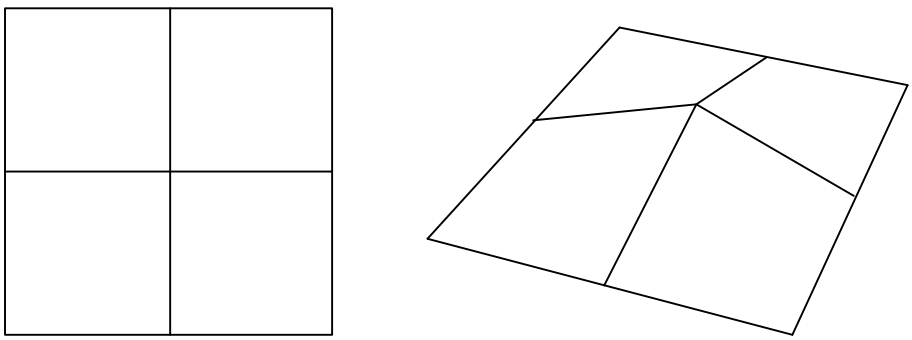
This will created your landscape at position (0,0,0). Note that by default, the size of the landscape is just one unit square. Following that, you must set the seed for the random number generator. This is a very important step - each seed creates a different landscape, but using the same seed twice should create the same landscape for both! Anyway, the seed is set using the `'setSeed()'` method.

```
island.setSeed(300); // seed can be any integer
```

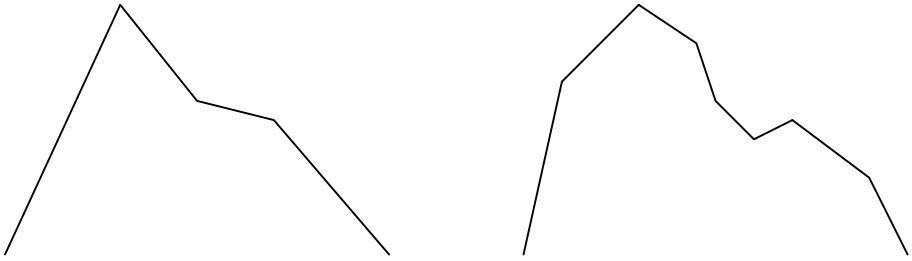
You should be aware that there is another way of creating the landscape which allows you to specify the seed at creation time:

```
Landscape3d island = new Landscape3d(seed,0,0,0);
```

What this gives you is a 3x3 array of points, connected to form 8 triangles. It looks a little like this:



So, this doesn't look much like a landscape, does it? What we need to do now is add more, random data. This is achieved by taking the height value between 2 points and adding a random amount to it. In 2 dimensions the effect would look like this:

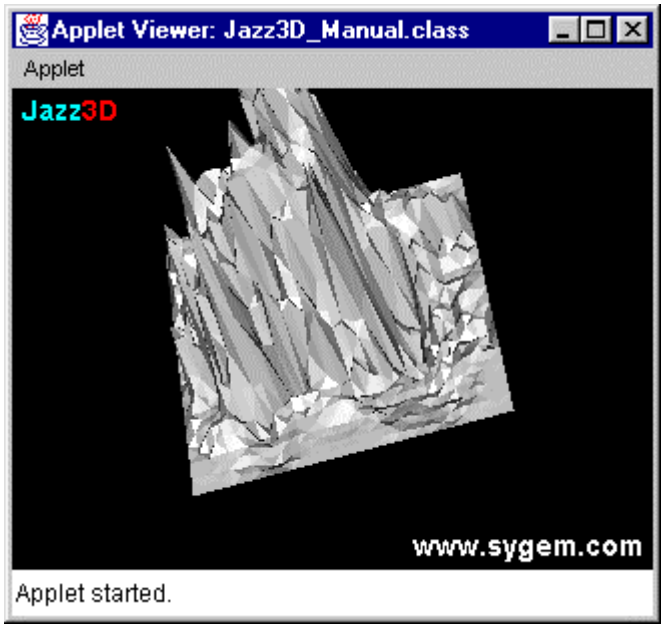


As you can see, more detail has been added to the landscape in a kind of random, kind of structured way. This method is called 'fractal sub-division'. To achieve this effect, we use the `'fractalize()'` method:

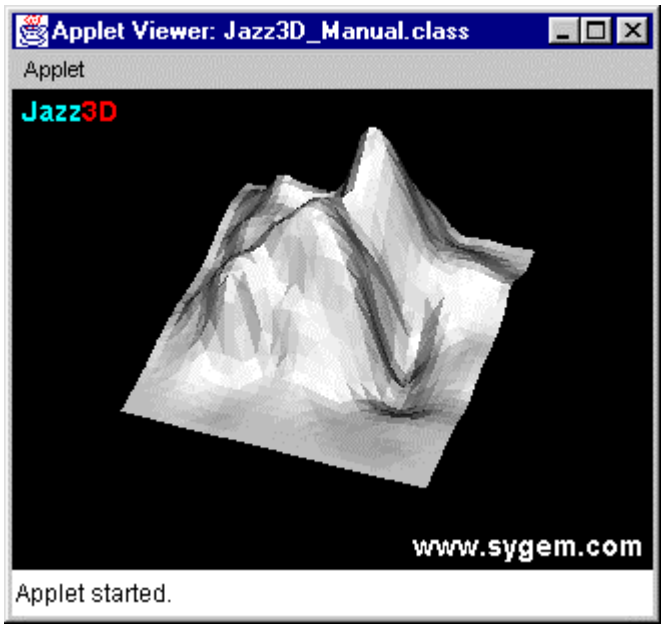

```
island.fractalize();
```

Please note that this can very quickly cause your landscape to get very detailed. Using `'fractalize()'` once expands the landscape from 8 (2 x 2 x 2) triangles to 32 (4 x 4 x 2) triangles. Doing it again will result in 128 (8 x 8 x 2) triangles, and once more would give 512 (16 x 16 x 2) triangles! All this in just 3 method calls.

So now what we have is a landscape consisting of 512 triangles, but as you can see on the next page, the landscape is very sharp and angular.



Fortunately, there is a way to correct this sharp nature of the landscapes. A simple call to the `'smooth()'` method will help greatly. You can use `'smooth()'` as many times as you like, but you loose a certain amount of detail on each call. It definitely improved the look of your landscape though. Here is the same landscape having had `'smooth()'` applied to it twice.



The final step necessary to create the landscape is to tell it go actually join the points and create the triangles. This is done using the `'generateShape()'` method. Here is a complete sequence for creating a landscape:

```
Landscape3d island = new Landscape3d(0, 0, 0);

island.setSeed(12345);

island.fractalize();

island.fractalize();

island.smooth();

island.fractalize();

island.fractalize();

island.smooth();

island.smooth();

island.generateShape();
```

But that's not the whole story. If that was all you could do with the landscape it would be a little dull, after all. So, here are a few more tricks you can apply to create more interesting, and more useable landscapes.

Coloured landscapes

Adding colour to your landscapes can make them so much more realistic - and it is so easy to do! The basic principle behind it is that you define a colour at a given altitude (between 0 and 1), and anything higher than that number is drawn using that colour. All faces start off white, so if you wanted all faces to be drawn in red (scary…), you could use this code:

```
island.addColour(0,255,0,0);
```

The first parameter specifies the altitude. The other 3 represent the R,G,B components of the colour. If you then wanted everything above half way to be blue, you could say:

```
island.addColour(0.5,0,0,255);
```

There is no limit to the number of different colours you can define for your landscape - although the more you add, the more confusing it can get! This approach also works very well with the gouraud shaded renderers, giving very nice changes in colour.

Stitch 'em up!

We got thinking that maybe having just one landscape in your world, with lots of detail, is not the most efficient way of handling landscapes. After all, the chances are that bounding-box checks won't work - this would mean that every face in the landscape must be drawn. What if you could define multiple landscapes, and stitch them together, a bit like a patchwork quilt? Well, you can. Basically, a method exists which will set all of the edge points in the landscape to zero. This gives them all a common edge which can then be placed next to another landscape, to form a bigger landscape. Using this approach you can split the landscape into bigger, more varied sections.

```
island.forceEdgesToZero();
```

Other methods

By default, the faces which make up the landscape are double sided - if you looked at the landscape from the bottom, you would still be able to see it. This is not always the most efficient way of drawing this kind of object, however. If you never want to see the underside, you might as well set the faces to be single-sided. Here's how.

```
island.setSingleSided(true);
```

If at any point in the landscape generation process you feel that the landscape is not random enough for you, you can add a random amount to every point in the current landscape. This will increase the jaggy look of your landscape, so you may need to smooth it afterwards. By default, the random values are in the range -0.5 to 0.5.

```
island.addRandomValues();
```

Or, you can specify the range the random values will take. This allows you to be a little more selective about the random values - you can specify a small range if the landscape is already very detailed, and you don't want to mess it up too much. For example, to specify a range of between 0.1 and 0.2, use this method:

```
island.addRandomValues(0.1,0.2);
```

Jazz3D actually comes with an applet that lets you explore the world of fractal landscapes, and generate code for you - it can be accessed from the Demos directory (a file called FractalExplorer.html).

Sprites

In version 2 of Jazz3D, there was a separate 'sprite3d' primitive. This primitive was designed to allow you to display a single bitmap image, mapped onto a single quadrilateral primitive, with a transparent colour to allow you to see through part of the image. However, this is no longer required in version 3.

To achieve this effect, you can use the following method:

- Create a Quad3d primitive
- Create a RenderTextured renderer
- Load a Texture
  - Set the transparent colour of the texture
  - Add the texture to the renderer
- Assign the renderer to the Quad3d
- Add the Quad3d to the World

And that's it. Because you have set the transparent colour of the texture, this will be picked up when the object is rendered. The advantage of this approach is that

you are no longer limited to a single Quad3d for the sprite effect. You can apply this to absolutely any object in your World.

Freeform objects

So far, all we have discussed are primitives which allow you to create relatively simple shapes with ease. However, you may wish to create a unique 3D object, one that cannot be created using just the standard primitives. This is where the freeform object comes in very handy. Basically, it allows you to define all of the points that make up your object, then stitch them together to make the actual polygons. Here's how it works:

Firstly, create your freeform object - all it requires is the initial position of its center:


```
Freeform3d obj = new Freeform3d(0,0,8);
```

Next, we define the points that make up our object. In this simple example, the 8 points which make up a cube are shown. Note that these co-ordinates are relative to the object center.

```
obj.addPoint(-0.5,-0.5,-0.5);
obj.addPoint(0.5,-0.5,-0.5);
obj.addPoint(0.5,0.5,-0.5);
obj.addPoint(-0.5,0.5,-0.5);
obj.addPoint(-0.5,-0.5,0.5);
obj.addPoint(0.5,-0.5,0.5);
obj.addPoint(0.5,0.5,0.5);
obj.addPoint(-0.5,0.5,0.5);
```

Note:

It is also possible to specify the UV mapping co-ordinates of each point in your Freeform object directly. There is another form of the ‘addPoint()’ method that takes 2 extra parameters - one each for the UV pair. But don’t forget to turn on the Vertex UV mapping system!

Next, we can define the sides of the cube. This is done using the `'addFace()'` method.

```
obj.addFace(0,1,2,3,255,255,255,false);

obj.addFace(4,5,6,7,255,255,255,false);

obj.addFace(2,6,5,1,255,255,255,false);

obj.addFace(7,3,0,4,255,255,255,false);

obj.addFace(0,1,5,4,255,255,255,false);

obj.addFace(7,6,2,3,255,255,255,false);
```

The first four numbers here represent the points you should already have added to the object. Number zero represents the first point you added. Number three is the fourth point, and so on.

The next three numbers let you specify the colour of the face. In our example the faces are all set to be white. This follows the standard Red, Green and Blue notation used throughout Jazz3D.

The final variable is a boolean flag which allows you to specify whether or not the face is to be double-sided. A double-sided face is not affected by 'backface culling', and will always be drawn if on screen. It can save you having to define 2 faces, however. Note that for a cube like ours, there is no point in using double-sided faces, because only one side can every be seen. This is the same for all solid shapes.

There is also a version of `'addFace()'` you can use to add triangular faces to your objects - just use three numbers instead of four.

Finally, you need to call the `'prep()'` method on your freeform object. This calculates various things about the object, ready for speedy rendering.


```
obj.prep();
```

Writing your own primitives!

Almost worthy of it's own section, we couldn't let you go without explaining how to create a primitive of your own. This can be really useful - imagine that you want your program to feature a certain type of shape over and over, but that shape doesn't exist in the provided primitives. You could write a method to generate a

Freeform3d object in the correct shape, but it would be much better if that shape could be an object in it's own right. So let's do it! Let's create a `'Disc3d'` primitive - a circular shape in 3 dimensions.

First steps

The first thing you will need to decide is which package you are going to place your primitive in. For the purposes of this example, we will put it in the same place as all the other primitives. So let's write a bit of code:

```
package com.sygem.jazz3d3.primitive;

import com.sygem.jazz3d3.Object3d;

import com.sygem.jazz3d3.Triangle;

public class Disc3d extends Object3d {

     public Disc3d(int sections,

                   double x,

                   double y,

                   double z) {

          super(x,y,z);

     }

}
```

Phew! This small piece of code actually gives us a large amount of what we need to create our disc. The important bits to note are:

- We are importing Object3d and Triangle from the core Jazz3D package
- We are extending the Object3d class for our Disc3d
- In our Disc3d constructor, we call the super class constructor, passing in the co-ordinates of the center of our object.

Of course, this doesn't give our object any points or faces, so it is by no means complete!

Create vertices

Let's create a method which will create a number of vertices for this object. The vertices need to be in a circle, around our center point, varying in the x and y dimensions. This will ensure that the disc is facing us by default. And don't forget the center point of the disc!

```
private void createDiscVertices(int sections) {
     addVertex(0,0,0,0); // Center vertex
     double theta = 180/sections;
     double angle = 0;
     double rad = Math.PI / 180.0;
     for (int i=1;i<sections+1;i++) {
        angle += theta;
        float x = (float)Math.sin(angle*rad);
        float y = (float)Math.cos(angle*rad);
        addVertex(i,x/2,y/2,0);
     }
}
```

Again, there are a few important things to note about what we have just written.

- The 'addVertex' method takes 4 parameters - the first specifies the position in the vertex array, used when we create faces later. The next 3 specify the position of the vertex, relative to the center of the object.
- You can specify the vertices in what ever order you want - whatever is easiest to code!

This method should now be called from within our constructor, after the call to `'super'`. And now we are ready to create the faces of out object.

Create faces

We will have another separate method for creating our object's faces. Take a look at the following code:

```
private void createDiscFaces(int sections) {
    Triangle t;
    for (int i=0;i<sections;i++) {
        if (i==sections) {
          t = new Triangle(0,i,1,255,255,255);
```

```
          addTriangle(t);
        } else {
          t = new Triangle(0,i,i+1,255,255,255);
          addTriangle(t);
        }
    }
}
```

That's all there is to it! Faces can be added as you like - but do make sure you are only referencing vertices that have already been created, otherwise when the object is being drawn an error will occur. You can specify the colour that each face should be, and basically perform any operations from the Object3d class on your primitive.

Again, this method must be added to our constructor, this time following the method which created our vertices.

Finish up

To finalize the creation of our new primitive, there are just 2 lines of code we need to add to our constructor. One of these will resize the vertex and face arrays to be the correct size, and the other will do some other useful stuff - including the creation of an optional bounding-box, used for object culling.

```
cleanUp();

prepForDisplay(true); // to create bounding-box
```

And that's it! Our `'Disc3d'` is complete - and here is the complete source code.

```
package com.sygem.jazz3d3.primitive;

import com.sygem.jazz3d3.Object3d;
import com.sygem.jazz3d3.Triangle;

public class Disc3d extends Object3d {

     public Disc3d(int sections,
                   double x,
                   double y,
                   double z) {
```

```
            super(x,y,z);
            createDiscVertices(sections);
            createDiscFaces(sections);
            cleanUp();
            prepForDisplay(true);
        }

    private void createDiscVertices(int sections) {
        addVertex(0,0,0,0); // Center vertex
        double theta = 180/sections;
        double angle = 0;
        double rad = Math.PI / 180.0;
        for (int i=1;i<sections+1;i++) {
            angle += theta;
            float x = (float)Math.sin(angle*rad);
            float y = (float)Math.cos(angle*rad);
            addVertex(i,x/2,y/2,0);
        }
    }

    private void createDiscFaces(int sections) {
       Triangle t;
       for (int i=0;i<sections;i++) {
           if (i==sections) {
             t = new Triangle(0,i,1,255,255,255);
             addTriangle(t);
           } else {
             t = new Triangle(0,i,i+1,255,255,255);
             addTriangle(t);
           }
       }
    }
}
```


Exercise:

Try typing this primitive in and creating a simple program to create loads of `'Disc3D'` objects.

Chapter 11

Model Loading

The best way to get realistic objects into your programs

Loader Objects

In the same way that the primitives have been placed in an external package, the loaders have received the same treatment. Again, the advantage of this is that if you don't want to use the loaders, you don't need to download them from the net.

```
import com.sygem.jazz3d3.loader.*;
```

The ability of Jazz3D to load objects created using 3d modeling packages opens up a whole new dimension to your programs. This excellent feature allows you to add truly unique, highly detailed models to your worlds. The following object formats are supported.

- GEO
- GEM
- ASC (3D Studio ASCII format)
- 3DS (3D Studio binary)
- NFF
- OBJ
- PLG
- LWO (Lightwave)

The first step to loading an object file is to create an instance of a 'LoadFactory'. This class will hold the loaders for all of the supported formats you choose to use.

```
LoadFactory lf = new LoadFactory();
```

Now all you need to do is register with the loader the file types you want to use. Let's say that we want to load a file called 'X29.asc'. We need to associate the extension ('asc') with the ASC loader. And this is how it's done.

```
lf.assignLoader("ASC", new LoaderASC());
```

Easy! The string representing the file extension can be in upper or lower case (it's all converted to lower case internally). How about loading a file called 'Dragon.geo'? The next 3 lines of code are all you really need to enable you to load models into your Jazz3D programs.

```
lf.assignLoader("GEO", new LoaderGEO());
```

This means that you can associate any file extension with any loader - useful if uploading files to somewhere which doesn't support that file extension. As you can probably guess, once the file loader has been assigned, any number of objects of that type can be loader. And this is how to do the actual loading. Firstly, we create a 'Model3d' object.

```
Model3d dragon = new Model3d(0,0,8);
```

Like many of the primitives, this just takes the position in space as it's parameters. All we have to do next is call the `'loadModel()'` method, passing in the filename and the LoadFactory object we just created.


```
dragon.loadModel("Dragon.geo", lf);
```

Providing the model file can be found, the object is loaded and you can add it to the world in the normal fashion. If the file is not found, an exception will be generated and output to the Java console. Your program will continue to execute, but the object you tried to load will not exist in your World.

Note:

At the moment, UV texture wrapping is only supported for models loaded using the 3DS loader.

Threaded loading

The problem with the above approach is that it freezes your program until the loading has completed. This makes it impossible for us to display anything on the screen whilst the object is loading. If you wanted the object to load before your program ran, it would make it awkward to display anything before the loading - how would users know that anything was happening?

Are there any solutions to this? Well, you could, in theory, create yourself a separate thread in which to do this loading and control the thread yourself. This would at least allow you to put something on screen before the loading was completed. However, you would have no idea about how much of the object had been loaded, so the display would be of little real use. That's why we came up with another way to load objects - the threaded approach.

First Steps

The first steps towards loading a model in a thread are actually the same as for the legacy approach. The Model3d is created, along with a LoadFactory object, and any loaders you want to use are assigned to the LoadFactory in the usual way. It is only after these steps that things start to differ. Instead of using the command

```
dragon.loadModel("Dragon.geo", lf);
```

to initiate the loading of the object, we instead use this command:

```
dragon.loadModelInThread("Dragon.geo", lf);
```

What does this do? Well, it doesn't actually do much, except for setting up the thread ready for the loading. It is up to you to decide when the loading takes place, so you are able to initialise the display as you like. And when you are ready to begin loading, you simply issue the following command:

Start the process


```
dragon.beginLoadingInThread(1);
```

In this case, the parameter represents the number of milliseconds the loading thread should pause for every time a line is read from the input file. The smaller the number, the faster the object will be loaded - but if you want to perform any other actions whilst the loading is taking place, then the number should be increased to let other threads perform their work as well.

Once the loading has begun, you will need some way of knowing when it has completed (obviously you don't want to add the object to the World before the loading is finished). There is an easy way of checking the load status, which also allows you to perform other operations whilst the loading is happening:

```
boolean stillLoading = dragon.loading();
```

If this method returns true, then you know that the loading has not yet completed. The best way to use this is to employ a while loop to continuously check the load status, and only continue once the loading has completed:

```
while (dragon.loading()) {
     // Perform any additional operations
}
```

Once the loading has actually finished, there is one more thing you need to do to the object before it can be added to your World. This additional step is required to ensure that the object renders correctly.

```
dragon.finishLoadingInThread();
objId = myWorld.addObject(dragon);
```

Load progress

So, what we have so far is a means of loading your 3d models in a separate thread - which is all very well, but not really that much different to the old system, because any display you put on the screen will not be able to inform the user of the progress of your load operation - just that it is still taking place. What we really need is a way of finding out how much of the load has occurred at any given point in time. Fortunately, Jazz3D provides just such a method:


```
int percentLoaded = dragon.getLoadProgress();
```

This method will return the percentage of the load that has completed so far (not very accurately, because it only uses integers, but that is enough for any display on screen). The great thing about this method is that it can be called from a method which draws it to the screen - like this:

```
void paint1(Graphics g) {
  g.setColor(new Color(255,0,0));
  int loadProgress = dragon.getLoadProgress();
  g.drawString(loadProgress+"%",20,80);
}
```

To call this whilst loading the actual object, we would need to modify the while loop introduced earlier to include a few extra commands. The complete code for loading a model in a separate thread and using Jazz3D to display the load progress on screen looks like this:

```
dragon.beginLoadingInThread(1);
while (dragon.loading()) {
     myWorld.prepareCanvas();
     myWorld.drawImage();
     paint1(myWorld.getCanvas());
     myWorld.finishCanvas();
}
dragon.finishLoadingInThread();
pid = myWorld.addObject(dragon);
```

Of course, you can use the load progress to produce any kind of display you like - load bars, circular load progress meters - the only limit is your imagination!

Exercise:

Write a program that will load in any 3d object from disk (or across the Internet), and display a fancy load bar whilst it is doing so.

VRML Loader

A late addition to the release of version 3 of Jazz3D was this - the VRML loader. It is a partial implementation of the VRML 97 standard - partial in that it doesn't implement a large subset of the commands. What it does give you is the ability to load the basic geometry contained within the VRML file.

The VRML loader is used in much the same way as any of the other loaders. However, because this loader is contained in a package all by itself, you will need to import both the loader package and the VRML package.

```
import com.sygem.jazz3d3.loader.*;

import com.sygem.jazz3d3.VRML.*;
```

Rather than use the 'Model3d' primitive, there is a new 'VRMLModel3d' class which must be used with the VRML loader. This VRML model class gives you a number of extra methods you can use to improve the viewing experience for the end user. For example, often you may find that the units used in the VRML model file are very large - much larger that a Jazz3D world unit. What you then need is a way of scaling the object down to a reasonable size on screen. That's where the `'scaleScene'` method comes in.

```
VRMLModel3d vrml = new VRMLModel3d(0,0,8);

.. .. // load model here..

vrml.scaleScene(2,2,2);
```

The `'scaleScene'` operation will make your object, in this case, 2 Jazz3D units in each direction. If you want, you can of course make it bigger or smaller along any of the 3 primary object axis. In this case, the aspect ratio of the object may not be preserved. Why? Well, if the object in question was a long cylinder, after the `'scaleScene'` it would be as wide as it is long - not the same object as before!

Fortunately, another version of the `'scaleScene'` method is provided which will allow you to keep the aspect ratio if the objects intact. All you need to do is pass in a single parameter, specifying the length in units of the largest side of the object.

```
vrml.scaleScene(2);
```

Another useful method that the VRML model gives you is the ability to center the scene on the co-ordinates you specified when creating the object. Again, sometimes the VRML model you are loading can have its center point somewhere other than the origin. When you display this on screen, the object may not even be visible! All you need to do is call this method to solve that problem.

```
vrml.centerScene();
```

Our VRML loader also gives you access to a few other nodes which are supported by the loader, but are not anything to do with the object geometry and are therefore not a part of the object itself. For example, lights and viewpoints will be loaded and made available to you using the following methods:

```
Camera3d[] vrmlCameras = vrml.getCameras();

Light[] vrmlLights = vrml.getLights();
```

These can the be added to your World in the usual way.

> Note:
>
> Any lights or cameras taken from the VRML loader will have their local co-ordinates translated into world co-ordinates, so beware.

So that's about it for the VRMLModel3d class, but there is still more to tell you - this time about the VRML loader itself. There are 3 things you may need to configure regarding the loading of VRML objects. Firstly, the path to the Jazz3D fonts. This is required if you will be loading a VRML scene that uses text.

```
LoaderVRML lv = new LoaderVRML();

lv.setFontPath("fonts"); // No trailing slash…
```

Another useful thing you can configure is the 'granularity' of the primitives created by the loader. By this, we mean the number of sub-sections used when creating things like spheres and cylinders.

```
lv.setPrimitiveGranularity(10);
```

Finally, you can also specify the size that any texture loaded by the VRML loader will be. This can help to avoid any scaling defects, by loading the textures larger than the

real image. Note though that for the textures to be loaded, a texture mapping renderer must be applied to the object BEFORE the object is loaded. Otherwise the textures will not appear.

```
lv.setTextureSizes(Texture.MEDIUM, Texture.LARGE);

lv.setTextureSizes(Texture.LARGE);
```

Future plans for Jazz3D include expanding the VRML support, hopefully to include the interpolators and possibly the sensors as well. Watch this space!

Loading Compressed Files

Jazz3D now features the unique ability to load any type of 3D object from within a ZIP or Gzip'ed file. The first thing you require for this feature is the following package:

```
import com.sygem.jazz3d3.decompression.*;
```

You will also require JDK1.1 or higher to use this feature.

Inside this package are 2 very small classes - ZipDecompressor and GZipDecompressor. To activate the decompression, you would use a piece of code a little like this…

```
Loader3DS myLoader = new Loader3DS(); // e.g.

ZipDecompressor zd = new ZipDecompressor();

myLoader.setDecompressor(zd);

myLoadFactory.assignLoader("3DS", myLoader);

// then load the object as normal!
```

The decompression is totally transparent to the end-user, and the object will be loaded as normal. There are, however, a few things to take note of.

1. The model file must be the first entry in the ZIP file.

2. For the threaded loading, the Gzip decompressor doesn't correctly return the file size, which results in some peculiar values.

Chapter 12

Advanced Techniques

Some cool stuff from Jazz3D…

Hierarchical Objects

Examples of a hierarchical object model can be seen in so many computer games these days. Any self-respecting 3D action game features characters whose limbs can move independently of each other, and of the body of the character. This is only really possible with hierarchical objects.

Hierarchical objects deal with the notion of an object having children. When these children are moved (rotation only), the parent object does not move. However, when the parent moves (as either a rotation or translation), all of the children go through the same movement. This gives the impression that the parent and children are attached.

Here's how...

So, the first thing you need to do is define the objects which are going to be the final set of children in the object. We will be using cubes of decreasing size in this example.

```
Cube3d obj3_1 = new Cube3d(1.1,0,8.15);
obj3_1.scaleObject(0.2,0.2,0.2);
Cube3d obj3_2 = new Cube3d(1.1,0,7.85);
obj3_2.scaleObject(0.2,0.2,0.2);
```

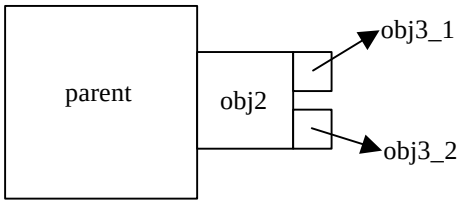
Now, we define the object which will be the parent of these 2 children.

```
Cube3d obj2 = new Cube3d(0.75,0,8);
obj2.scaleObject(0.5,0.5,0.5);
```

And finally, we will define the parent object.

```
Cube3d parent = new Cube3d(0,0,8);
```

So, that gives us 4 cubes, of differing sizes, and would look like this from above (the positions of the cubes are still set when creating the object):



The only problem with this is that the objects are not connected in any way. If you moved `'parent'` in any direction, the other objects would still not move. So what we need to do is attach the children to their parents, like so:

```
int obj3_1_id;

obj3_1_id = obj2.addChild(obj3_1, 1.1,0,8.15);

int obj3_2_id;

obj3_d_id = obj2.addChild(obj3_2,1.1,0,7.85);

int obj2_id;

obj2_id = parent.addChild(obj2,0.75,0,8);
```

The children are now associated with their parent. To add this object to your Jazz3D world, all you need to do is add `'parent'`, like this:

```
int parent_id = myWorld.addObject(parent);
```

So how did we come up with those magic numbers in the `'addChild'` commands? Well, the key thing to note is that these co-ordinates represent the pivot point that the child object will be rotated around, and that they are relative to the position of the child object itself. Let's take a closer look at the final parent-child association above:

The object `'parent'` has it's center point at (0,0,8), and the size of the object is (1,1,1). This means that the right edge of the object (in the diagram above) has an x-value of 0.5. The object `'child2'` is at position (0.75,0,8) and has a size of (0.5,0.5,0.5). That means that the left-

most edge of the object also has an x-value of 0.5, so the 2 objects are touching each other. When the child object is added to the parent, we gave it a value for the pivot-point of (0,0,0). That means that when you rotate this child object, it will rotate around it's own center. If you wanted it to rotate around the center point of the parent object, you would give the pivot point as (-0.75,0,0) - the pivot point is always given as a point relative to the object center.

Note:

All the child objects use the same renderer as the parent - it is not possible to use different renderers, but all renderers are supported. Nothing special needs to be done for drawing the object. Jazz3D automatically detects if the object has any children, and draws them straight away.

And that's it! Now when you translate or rotate `'parent'`, the children move along with it.

Child rotation

OK, so you can now get sub-objects moving and rotating along with their parents. However, this is only half the battle - wouldn't it be nice if you could rotate the sub-objects as well, to allow for truly independent movement? Well, of course you can, and here's how to do it.

All you need to do is get a handle on the sub-object in the same way as we have seen previously - all you need to know is the integer ID returned when you first added the object to its parent. For example:

```
mySubObj = myWorld.getObject(child2_id);
```

Don't forget that you can also make use of the other version of `'getObject'` which looks for objects based on their name:

```
child2.setName("Sub Object 1")

…

mySubObj = myWorld.getObject("Sub Object 1");
```

Once you have your sub-object, you can use any of the usual methods of an object, including the rotation methods. You should use the `'rotateLocal'` methods for rotation, because the rotation needs to follow the axis of the object, rather than the world axis.

Beware:

Translations can also be applied to the sub-object - however, that will cause the objects to lose their positions relative to each other. Be careful which operations you apply to your sub-objects, and try to stick to rotations wherever possible.

Fogging

Fog provides an easy way of giving your world a more realistic feel. After all, fog is a real world phenomenon - why not make your world feel as real as possible.

Creating Fog

The fog is turned on and off with one simple method call - `'setFog()'`. Like this:

```
myWorld.setFog(true); // will turn fog on

myWorld.setFog(false); // will turn the fog off
```

Fog distances & colour

The fog is fully configurable, as you would expect from Jazz3D - although there are only 3 things you really need to configure! This makes providing fog very straight-forward.

The first thing you can set is the distance from the viewpoint where the fog starts. This can be any number, as long as it is greater than zero (if you use a negative number, Jazz3D just makes it zero for you). This is all set through one method call:

```
myWorld.setFogStart(double distance);
```

And setting the distance to the end of the fog is just as easy. The end of the fog is actually the point at which the fog appears 100% dense (i.e. you can't see any of your objects - just fog). Use the following method:

```
myWorld.setFogEnd(double distance);
```

You can also change colour of the fog - it doesn't have to be grey, you know! A simple call to the following method will change the colour of the fog:

```
myWorld.setFogColour(0,0,0);
```

Note that by default, the fog colour is BLACK. This gives an interesting effect, known as 'depth-cueing', used to accentuate the perception of depth in a 3D engine. However, it doesn't look like much like fog! So, for example, if you wanted a red fog (very menacing), you would use the following lines of code:

```
myWorld.setFogStart(5); // The min fog distance
myWorld.setFogEnd(30);  // The max fog distance
myWorld.setFogColour(255,0,0); // RED fog
myWorld.setFog(true);   // Enable the fog
```

Once the fog has been created and turned on, you can modify the distances and colour at any time in your world.

Performance Issues

When an object is not in the fog, there is no performance hit associated with having fog present in your world. Similarly, if an object goes past the end of the fog,

Note:

If you use fog, you will not be able to use a background image - because the fog obscures it! You will also not be able to set the background colour, because the background colour becomes the same as the fog colour.

You should also note that since no object can be seen past the end of the fog, you can set the value of the Yon plane (see below) to be the same as the end value of the fog. This will make your world as efficient as possible.

Scene Management

Jazz3D now gives you the ability not only to create objects on-the-fly and add them to your world, but also the ability to delete objects as well. This makes Jazz3D worlds more memory conscious than they have previously been, and allows you to create simulations that can manage their objects more effectively, and more efficiently too.

Deleting Objects

To delete an object is very simple. All you need to do is call the `'deleteObject()'` method of your world, passing in the integer variable returned when you added the object. For example:

```
Cube3d cube = new Cube3d(0,0,8);

int i = myWorld.addObject(cube);

myWorld.deleteObject(i);
```

And like most of the other methods of this kind, there is a corresponding version to allow you to delete objects based on their name:

```
myWorld.deleteObject("Object 1");
```

The object will not be drawn any more. Note that this is final, and there is NO undo! It is also now possible to delete all objects from your world with just one method call. This can be especially useful if you want to define a completely new scene, but keep the same world object. Once again, there is no undo for this feature, so use it with some caution!

```
myWorld.deleteAllObjects();
```

Adding Objects at Runtime

It is also possible to add new objects to your world during the execution of your program. The method for adding the objects is the same as usual - `'addObject'` - the only difference is that you will need to call the following method to ensure that the object gets setup correctly for rendering:

```
myWorld.prepNewObjects();
```

Potential problems

Deleting objects or modifying objects whilst your program is running can lead to potential problems. For example, imagine that you re-generate your fractal landscape at some point, not in your main program loop (in response to a user action). This removes all vertex and face information from the object and re-creates it. But what happens if your main loop is still running? Jazz3D may try to rotate or redraw the object, before the vertices and faces have been recreated. The result would be an exception is generated, and your program stops.

So, to prevent this from happening, a set of methods are provided which will prevent Jazz3D from being able to access objects - `'suspend()'` and `'resume()'`. Here's how to use them:

1. Before making any changes to your objects, call `'myWorld.suspend()'`
2. Note that this will wait until the world has finished drawing the current frame
3. Then, you are free to change your objects as you want
4. Once you are done, call `'myWorld.resume()'`, and at the next frame, redrawing will commence.

Performance Issues

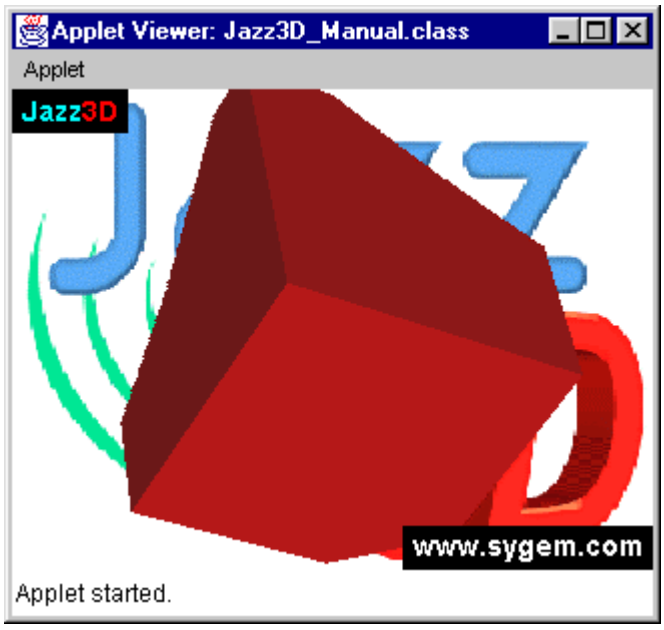
As mentioned before, deleting an object doesn't incur a real-time performance hit. However, once an object has been deleted, a space appears in the container used for storing objects. If you then try to add an object to your world, Jazz3D will try to add the object into these gaps (to keep the array under control). This will cause a minimal performance hit, depending on the number of objects already in the scene. However, this should be negligible compared to the creation of objects, which can be a much greater issue.

Hither / Yon Plane

The "Hither plane" is the imaginary plane in space where we perform near-z clipping. By default, this line is set to "z=0". What this means to you is that any object or part of an object that is behind the viewpoint is not drawn (which is good, because most of us can't see behind us). By changing the values of this, you can achieve some weird effects - such as the ability to see behind you as well as in front. In practice, however, the Hither plane rarely needs altering.

It is more useful to alter the Yon plane - for two reasons. The first is related to the way Jazz3D handles its z-buffer (the way it determines which objects are in front of other objects). Specifying a smaller value for the Yon plane means that the z-buffer will get cleared less regularly - but don't worry, you won't lose accuracy by doing this, you just won't be able to see objects which are very far away.

The other useful reason for wanting to change the Yon plane is to give the illusion of a solid back wall in your world - the image below is an example of this. The background is set normally (with `'setBackground()'`), and the cube is added at position (0,0,8). The Yon plane is then also set to 8. This means that any point with a z-value of greater than 8 will not be drawn, so the impression is that the background is a solid wall.



And finally, you need to know how to access this cool feature! It's very easy - just one method call:

```
myWorld.setHitherYon(double hither, double yon);
```

Performance Issues

The only bearing changing the hither / yon plane has on performance relates to how the z-buffer is cleared. We don't want to go into too much detail, suffice to say that the further away you set the Yon plane (i.e. larger values of z), the more often the z-buffer will be cleared - which causes a slight performance hit. Still, unless you set the Yon plane so far away that the z-buffer needs clearing every frame, you shouldn't notice any real difference in performance.

Chapter 13

Collision Detection

How to detect collisions between objects in your Jazz3D world

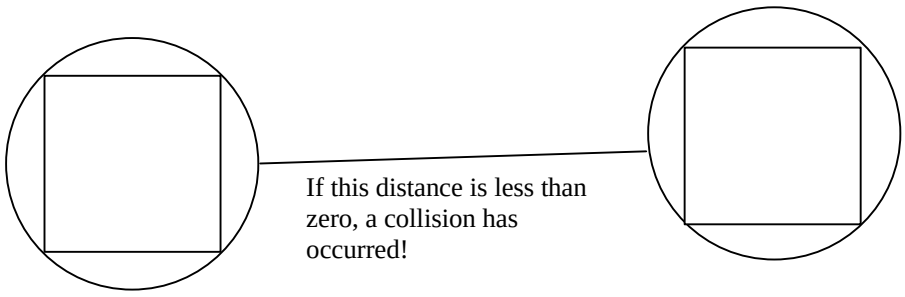
Simple Collision Detection

This is undoubtedly one of the most powerful features in Jazz3D - collision detection gives you the ability to find out if one object has collided with another. It's another step in creating a truly interactive world.

So how does it work? Well, there are just a few important methods you will need to be aware of for collision detection. The simplest method is to test for a collision between two objects.


```
boolean collide;

collide = myWorld.testCollision(obj1Id, obj2Id);
```

Given two object id's this will test to see if the objects in question are in collision. If they are, it will return TRUE, otherwise it will return FALSE. The collision tests performed are actually done with respect to a 'bounding sphere' around each object. Here is an example to illustrate this concept.



So, given this, we can also test on a single object, testing for collision with all other objects in your world!

```
int[] collisions;

collisions = my_world.testCollisionAll(objId);
```

As you can see, `'testCollisionAll()'` returns an array of integers representing all of the objects currently colliding with `'objId'`. If no objects are currently in collision with this one, `'collisions[0]'` will contain the value minus 1.

It is also possible to test for collisions with the current camera object. By default, the bounding sphere around a camera has a radius of 0.5 world units. This can be changed with a single method call.

```
myWorld.setCameraRadius(int radius);
```

With a camera radius, we can now test for collisions between objects in much the same way as before.

```
boolean test;

test = myWorld.testCameraCollision(objId);
```

This allows you to test for camera collisions against a single object. To test for camera collisions against all objects, you would use the following method:

```
collisions = myWorld.testCameraCollisionAll();
```

Again, this returns an array of integers, the first value being -1 if no collisions were reported.

Accurate Camera Collisions

The camera collisions can be set to use either the same spherical mode of collision detection as described above, or a more accurate polygon based mode. In this extra mode, the position of the camera (combined with the radius) is checked against the position of each vertex of the object. To set the camera collision mode, use the following code:

```
myWorld.setCameraCollisionMode(SPHERE_COLLISION);

myWorld.setCameraCollisionMode(POLYGON_COLLISION);
```

It is also possible to query the current camera collision mode:

```
int camMode = myWorld.getCameraCollisionMode();
```

Performance Issues

There is no noticeable performance hit when using the tests for collision with single objects. However, if you have a large number of objects in your world, you may notice a degradation of performance using the 'All' tests, which obviously need to perform their tests multiple times.

Accurate Collisions

The problem with the spherical collision detection is that is it a little too inaccurate, and those inaccuracies can be amplified depending on the shape of your object. Imagine that you wanted to perform collision detection on a missile shaped object - the trouble with an object like that is that it doesn't fit a sphere very well. This makes the collision tests very inaccurate.

What we really need is a more accurate way to detect collisions, whilst still retaining some of the speed of the spherical model. The best approach in this instance is to use a form or ray-casting. Basically, we can cast a ray (a line in 3d space between 2 points), and easily and accurately determine if a polygon is in collision with that ray.

The Test

This accurate collision mode comes down to just a single method call! Here is an example of it:


```
boolean test;

Vertex v1 = new Vertex(0,0,0);

Vertex v2 = new Vertex(0,0,1);

test = myWorld.pick(v1, v2, objId);
```

What we are doing is specifying the start and end points of the ray (represented as Vertexes), and then saying which object we want to test collision against, using the object ID returned when we initially added the object to our World. If there is a collision between the ray and any of the polygons of the object, the method will return true.

HitPoint

The HitPoint class is the key to this whole accurate collision thing. If the result of the accurate collision test is true, then the HitPoint class will have been filled out with all the relevant information regarding that collision.

HitPoint itself cannot be instantiated - it is a static class, and all the methods are static. This means that only information about only one collision at a time can be stored within the Jazz3D system. Once you know that a collision has taken place, you should use the following methods to retrieve the information about that collision:

To find out which object was involved in a collision (useful if you are iterating though a list of objects):

```
int objID = HitPoint.getObjectId();

String objName = HitPoint.getObjectName();
```

To find out which face on the object was involved in the collision (after which you can retrieve the actual face from the object if you want):

```
int faceID = HitPoint.getFaceId();
```

To get the exact position in space where the collision occured:

```
Vertex pos = HitPoint.getVertex(); // Or use…

double xPos = HitPoint.getX();

double yPos = HitPoint.getY();

double zPos = HitPoint.getZ();
```

To find the position along the ray where the collision occurred (for example, if the ray is 1 unit long and the collision happened three quarters of the way along the ray, this method will return the value 0.75):

```
double t = HitPoint.getT();
```

The next set of methods allow you to easily retrieve the face normals of the face being collided with (this information could also be obtained by getting the face directly and querying that):

```
Vertex normal = HitPoint.getNormalVertex();
```

```
double normalX = HitPoint.getNormalX();
```

```
double normalY = HitPoint.getNormalY();

double normalZ = HitPoint.getNormalZ();
```

The final operation you can perform on the HitPoint is to enable or disable the recording of that face normal operation:

```
HitPoint.setRecordNormals(true);
HitPoint.setRecordNormals(false);
```

Example uses

There are many, many ways in which this system can be used effectively. Here is one such approach:

- Add a Line3d primitive as a sub-child of your main object. This Line3d should represent the ray you want to test. For example, if you want to test forward moving collisions, you would create the Line3d pointing forward from the center of the object.
- At runtime, get the Line3d object back, and use the 2 vertices as the start and end point of the ray. Also, specify the object ID in the method call.
- If a collision is detected, you can then look into the HitPoint class for the information.

Potential problems

Don't forget that only one collision can be detected at any one time. Once another collision has been detected, the information stored in HitPoint has been overwritten.

You will also need to bear in mind that using this more accurate system of collision detection will be more computationally expensive than using the more simple spherical collision detection model (because every face in the object is being tested). To get around this problem, you may want to consider performing a simple test on each object in turn, and only doing the more accurate test if the spherical test is successful.

Chapter 14

VisiMagik

Jazz3D's real-time Image Processing subsystem

Introduction

VisiMagik was created to provide you with a 2D, real-time, image post-processor. What this means to you is that Jazz3D creates the image on the screen, then VisiMagik can take that image and process it. In addition to this, with Jazz3D version 3, we have extended VisiMagik to be able to handle textures as well as the final Jazz3D image.

VisiMagik has been designed to be separate from Jazz3D - all you need is an application which can generate an array of integers representing the display (this is what VisiMagik works with internally) and accept the modified array back again.

Create VisiMagik

Before you can use the VisiMagik system, you will first need to let Java know that you are going to use it - this is done using the import command, right at the top of your program (where you should already have imported `com.sygem.jazz3d3`).

```
import com.sygem.visimagik.*;
```

So, the first step to real-time image processing is to create the Visimagik object. This takes no arguments, so it's a really easy object to create:


```
Visimagik vm = new Visimagik();
```

Create and Assign Image Processors

Once the main VisiMagik system has been created, we get to create the processors we want to use. Here is a list of the currently supported image processors:

Image Processor name	Description
FilterGreyscale	Creates a greyscale version of the image
FilterMask	Masks out a selected portion of the image. This is defined by another image - GIF or JPEG. Any portion of this image which is not black is removed from the input image.
FilterNegative	Negates the contents of the image buffer. For example, (0,0,0) becomes (255,255,255). (132,15,67) would become (123,240,188).
FilterTint	This filter allows you to increase or decrease the values of the red, green and blue components of the image, up to a maximum of 255.
FilterEmboss	The classic image processing emboss feature…
FilterPastel	Gives the impression that your image was drawn with pastels
FilterThermo	Kind of a heat-seeking effect
FilterMotion	Creates a motion blur effect on your screen
FilterPlasma	Causes the image to wave about, like a flag
FilterGlass	Really nice glass effect
FilterGpen	Black and white pen style filter
FilterFader	Loads an image, either tiled or stretched, and fades between that image and your Jazz3D scene

Example constructors

The constructors for some of these filters are a little difficult to explain, so here are example constructors for each filter. Some of them are very simple, some of them less so.

```
FilterGreyscale grey = new FilterGreyscale();

FilterMask mask = new FilterMask();

FilterNegative neg = new FilterNegative();

FilterEmboss emboss = new FilterEmboss();

FilterPastel pastel = new FilterPastel();

FilterThermo thermo = new FilterThermo();
```

```
FilterMotion motion = new FilterMotion();
FilterPlasma plasma = new FilterPlasma();
FilterTint tint = new FilterTint(255,0,0);
// Will make every pixel's red part equal 255!
FilterFader fader = new FilterFader(
                    tx.getTextureArray(),
                    tx.getWidth(),
                    tx.getHeight(),
                    FilterFader.STRETCHED);
                    // where tx is a Jazz3D
                    // texture object.
```

Extra filter methods

Some of the filters have extra methods which allow you to tweak the behaviour of the filter either during run-time, or in a more flexible than using constructors. For example, `'FilterFader'` allows you to set the fade amount (how much the other image shows through above the Jazz3D image) at run-time. This is done through the static method `'setFadeValue()'`, passing in an integer between zero and 100. Here's an example of how to use this:

```
while (true) {
     FilterFader.setFadeValue(35);
     myWorld.redraw();
}
```

The other filter which offers you extra methods is the plasma filter. Here you have access to 5 methods to change the way the plasma works. The first of these is the `'setSpeed()'` method. It takes 2 integer parameters, one to set how quickly the plasma moves in the horizontal, and one to set the vertical speed. Smaller values will result in plasma which is slower moving (and arguably more pleasant…).

Next up is the `'setAmplitude()'` method. Again, this takes 2 integer parameters, but this time they affect the amount the image is shifted in the horizontal and vertical. Smaller values mean the image isn't changed too much.

Next, we have the `'setPhase()'` method. Once more, 2 integer parameters are used, and these specify the amount moved along the sine wave for each scan-line, or pixel. Related to this, you can also set the initial phase used - `'setInitialPhase()'`, and this also takes 2 integer parameters.

Finally, it is possible to set the colour of the background - this colour can be seen "behind" the source image, as a result of moving the image around. It is set using the `'setBackgroundColour()'` method, passing in 3 integers to represent the red, green and blue components of the colour.

Assigning the filters

Once your filters have been created, you can add them to the VisiMagik object, ready for running later. This is done in a very similar way to adding objects to your Jazz3D worlds. Here is an example:

```
int filter1 = vm.addFilter(emboss);

int filter2 = vm.addFilter(motion);
```

As with Jazz3D, these integer variables should be stored, as they will enable you to run your filters at a later date.

Removing filters

At any time, it is possible to remove a filter from the VisiMagik container. All you need to do is call the following method, passing in the integer returned when you added the filter in the first place.

```
vm.removeFilter(filter1);
```

Any attempt to run the filter once it has been deleted will simply be ignored.

Prepare for display

In the same way that your Jazz3D world needs to be prepared for finally rendering the images (using the `'prep()'` method), VisiMagik needs to be prepped too. This important step allows each of the filters to do any setup required (image scaling etc.), before run-time, to allow for maximum speed.

Here's how to perform this prepping:

```
vm.setDimension(myWorld.size());
```

Note that this must be done at the same point in your program as calling the `'prep()'` method of your Jazz3D world.

The Mask

Every filter can have a mask associated with it - what this means is that the filter only gets applied to certain areas of the screen. This can lead to some cool effects.

The mask itself can be any image, of any size. VisiMagik will make sure that internally it is scaled to the correct size. The mask works by only applying the filter to pixels where the mask image is BLACK. This is very important - if your mask image contains no black, the filter will not be applied at all.

To set-up a mask to use with your filter, it is easiest to use the texture loading facilities of Jazz3D. So, the first step would be to create and load yourself a texture.

```
Texture t = TextureLoader.loadTexture("a.gif");
```

Once this is done, you are ready to assign the texture to your filter. Just use the following method call:

```
filter.setMask(t.getTextureArray(),
               t.getWidth(), t.getHeight());
```

Basically, `'setMask()'` needs to be given an array of integers (representing the mask image), and 2 further parameters giving details of the width and height of the mask. And that's it! Your mask will now be active for that filter.

Of course, to use the Mask filter effectively, you need to assign a mask to it, otherwise nothing will happen!

Running Image Processors

By now, you should have your Visimagik container created, and have some filters assigned to it - all you need to do now is run them! Here is a sample main program loop which takes the image generated by Jazz3D, loads it into VisiMagik, runs a few filters, then passes the image back to Jazz3D for display.

```
while (true) {
  myWorld.prepareCanvas();
  myWorld.rotateObjectLocal(pid,0,y,0);
  myWorld.generateImage();
```

```
  vm.setImage(myWorld.getImage(),
              myWorld.size().width,
              myWorld.size().height);
  vm.runFilter(filter1);
  vm.runFilter(filter2);
  myWorld.setImage(vm.getImage());
  myWorld.drawImage();
  myWorld.finishCanvas();
}
```

So, there are a few important things to note here: firstly, rather than use `'redraw()'`, we are using `'generateImage()'`. These methods do basically the same things, but only `'redraw()'` actually draws the final image to the screen.

Following that, we need to get the image from Jazz3D, and into VisiMagik. The Jazz3D world has a method which can return the contents of it's image buffer, so that's what we shall use. The code looks like this:

```
vm.setImage(myWorld.getImage(),
            myWorld.size().width,
            myWorld.size().height);
```

Once that has been done, we are ready to run the filters we have previously added to the VisiMagik container. As you can see from the example above, this is done using the `'runFilter()'` method, passing in the integer you got when the filter was added. This approach gives you total flexibility about the order your filters are run in. If you don't can about the ordering of the filters, there is an alternative method you can use:

```
vm.runAllFilters();
```

As the name suggests, this method will execute all filters currently registered, in the order they were added to the VisiMagik container.

The final 2 steps involve returning the filtered image back to your Jazz3D world, then displaying this image on the screen. This is done with the following two lines of code:

```
myWorld.setImage(vm.getImage());

myWorld.drawImage();
```

And that's all there is to it. Your newly filtered image will appear on screen in all it's glory.

VisiMagik and Textures

VisiMagik has now been extended to work with Jazz3D Texture objects as well as being able to take the final generated image. This means that you can apply filters to individual textures before they are used in a texture mapper, or applied to the background of your Jazz3D program.

The idea behind it is very similar to the way VisiMagik works with the final generated Jazz3D image. There are 2 major differences when working with Textures - firstly, and fairly obviously, you need to get the information from a Texture object, rather than from the World. The other change is that you will need to run the image processors before you generate the final image, to ensure that the processors get applied to the Texture image before they are used in any texture mapper.

Here is an example of how to extract and put back the image information from a Texture:

```
vm.setImage(tx.getTextureArray(),
            tx.getWidth(),
            tx.getHeight();
// Run image processors
tx.setTextureArray(vm.getImage());
// Generate final image
```

Screenshots

To give you an idea of what each of the filters looks like, here are some screenshots of each filter in action - although you can't really get a feel for how they work from a static image - the only way to really appreciate them is to buy the full version of Jazz3D!

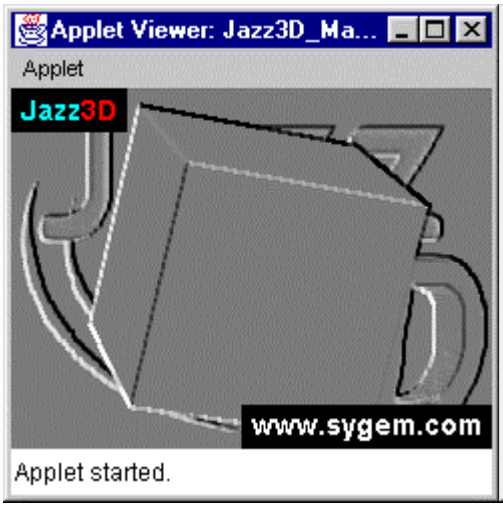
Greyscale

Mask

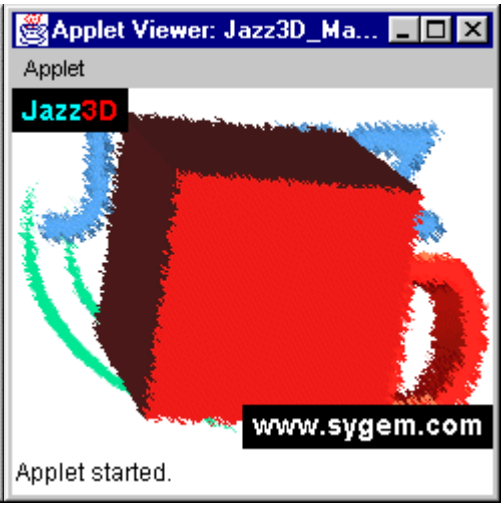
Negative

Tint

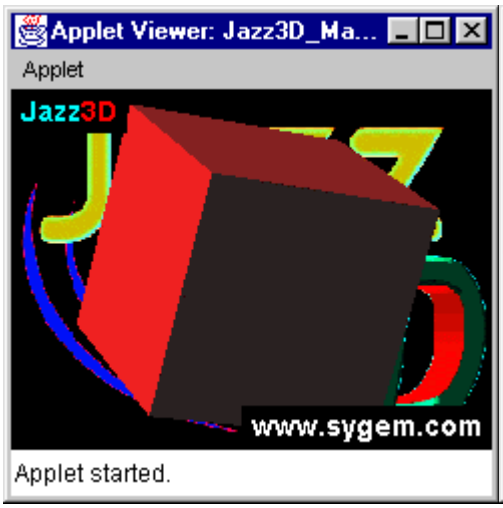
Emboss



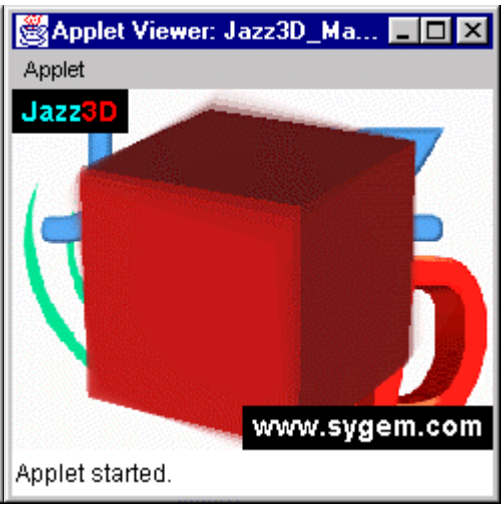
Pastel



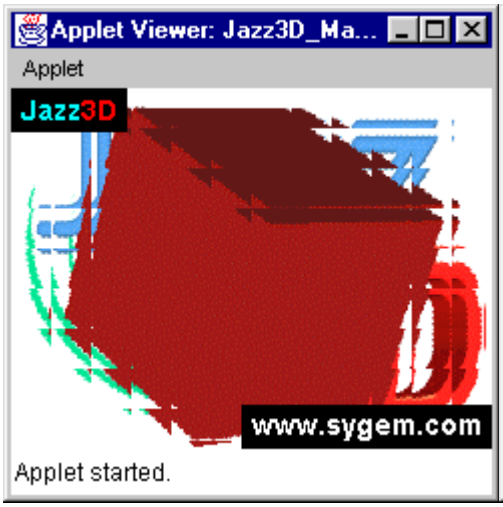
Thermo



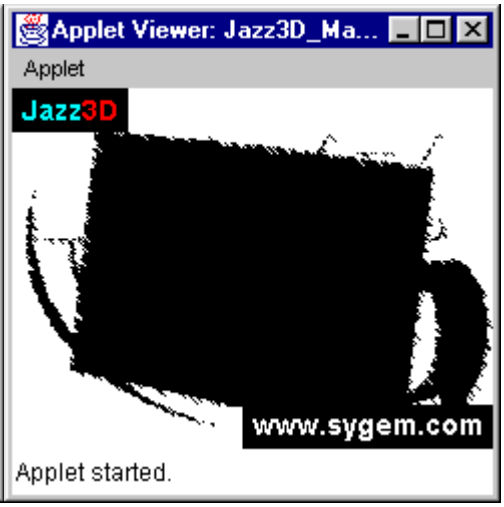
Motion



Glass

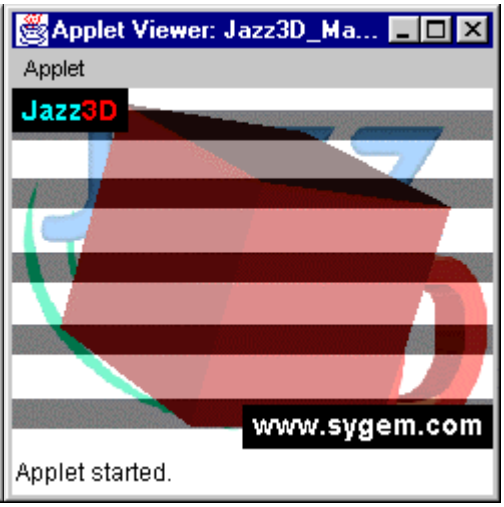


Gpen



Plasma

Fader



Chapter 15

An Example Program

The best way to learn is by example

So, now you've read through the whole manual, and are none the wiser. Had a look at the example programs and they all seem complete gibberish? In that case, let's take a step by step look at a simple example program - an applet with one rotating sphere and one light source.

First Steps

The first thing we need to do is tell the applet that it will be using the Jazz3D API - this is done using the import command, at the very top of the program, like this;

```
import com.sygem.jazz3d3.*;
```

We also need to import the AWT and Applet class, otherwise we can't create an applet.

```
import java.awt.*;
import java.applet.*;
```

We also need the basic framework of an applet. That means the class definition, and default constructor.

```
public class testSphere
extends Applet implements Runnable {
	public testSphere() {
	}
}
```

Hopefully this is all standard stuff and shouldn't hold any surprises for you. If it does - you need to learn some Java, quick!

Global variables

This simple program will require just 4 global variables;

Class	Variable name	Purpose
Thread	myThread	The thread object is required to get the program running - it also provides us with the ability to
World	myWorld	This is the Jazz3D world - the focal point of the program
int	objId	An integer variable which will be used to reference the sphere we will add to the world
int	lightId	Another integer which can be used to refer to the light source we are going to add to the world

Creating the world

Next, we will add all the code for creating the Jazz3D world, the sphere object and the light source. We will also see them bound together. All of this code goes in the `'init()'` method of the applet.

So, to create a world, we just create it like any Java object, with `new`. The parameter the world class takes in it's constructor is an Applet - since we are already in the applet's `'init()'` method, we can refer to it as `'this'`.

```
myWorld = new World(this);
```

Because the world class is actually an extension of an AWT Canvas, we need to add it to the layout manager of the applet. This is done in the same way as for any other AWT component, using `add()`;

```
add(myWorld);
```

OK - so now we have our world, and have added it to the applet. Now we should create the renderer object which we will use for our sphere. I think we will have it as a Gouraud shaded sphere - it's created like this;

```
RenderSolid gouraudShader = new RenderSolid();
```

Excellent. Now we can create the sphere itself. The constructor takes 5 parameters - 2 integers for the 'resolution' of the sphere, and 3 floating point numbers for the position in space (X, Y & Z). Refer to Chapter 5 for more details of this. Our sphere will have 10 vertical and 10 horizontal subsections - the gouraud shader allows you to use less faces, because of the smoothness of the results. The sphere will be at co-ordinate (0,0,5).

```
Sphere3d sp1 = new Sphere3d(10,10,0,0,5);
```

Next step is to tell the renderer we just created that it will be using gouraud shading, and then we need to associate the renderer with the object.

```
gouraudShader.setDrawingMode(Render.GOURAUD);

sp1.setRenderer(gouraudShader);
```

And now, we must add the sphere to the world. This gives us the value we can use later to refer to the sphere through the world.

```
objId = myWorld.addObject(sp1);
```

All we need to do now is create our light source and add that to the world as well. For this simple example, we will just use a directional light source - the simplest available. Refer to Chapter 6 for more on lights. Our light source will shine in the direction (0,0,1), which means directly into the screen.

```
Light tempLight = new Light(0,0,1);
```

And we add the light to the world in much the same way as we added the sphere. This also returns an integer value we could use to modify the light at run-time.

```
lightId = myWorld.addLight(tempLight);
```

And that's about it! At the end of all that, our `'init()'` method should look like this;

```
public void init() {
   myWorld = new World(this);
   add(myWorld);
   RenderSolid gouraudShader = new RenderSolid();
   Sphere3d sp1 = new Sphere3d(15,15,0,0,5);
   gouraudShader.setDrawingMode(Render.GOURAUD);
   sp1.setRenderer(gouraudShader);
   objId = myWorld.addObject(sp1);
   Light tempLight = new Light(0,0,1);
   lightId = myWorld.addLight(tempLight);
}
```

Threads

Remember that global variable called `'myThread'`? Here's where we add some code to deal with it. These are fairly standard Java methods for Applets - explanations of them can be found in any decent Java book.

```
public void start() {
     if (myThread == null) {
         myThread = new Thread(this);
         myThread.start();
     }
}

public void stop() {
     if (myThread != null) {
         myThread.stop();
         myThread = null;
     }
}
```

Run-time object manipulation

All we need to finish off our program now is the main program loop. Because we have written this as a threaded applet, this should be placed in the `'run()'` method.

The first important line of code we need is to call the `'prep()'` method of the world. This creates all the internal display variables, sets up the z-buffer and initialises loads of stuff. All you need worry about is that is gets called before you try to draw the world.

```
myWorld.prep();
```

Before we begin the main loop, we will also need some variables to store the angles of rotation for our sphere. These should be doubles, and I will call them x, y & z. Now we can enter the main program loop. This will be a simple while loop, which will never exit.

```
while (true) {
}
```

And it is inside this infinite loop we add the code to rotate our sphere. Because we are now in our run-time loop, we can only access the sphere through the world class. All we need to do is initialise the values of x, y & z before we enter the loop, then rotate the object, using the `'objId'` variable as a pointer to the sphere.

```
z = -1;
y = 3;
x = 2;
Object3d myObj = myWorld.getObject(objId);
myObj.rotateLocal(x,y,z);
```

The final step is to force the world object to update it's display. This done with the following method call;

```
myWorld.redraw();
```

And so the `'run()'` method will look like this;

```
public void run() {
   double x,y,z;
   x = 2;
   y = 3;
   z = -1;
   myWorld.prep();
   while (true) {
     Object3d myObj = myWorld.getObject(objId);
     myObj.rotateLocal(x,y,z);
     myWorld.redraw();
   }
}
```

The finished article

Here it is - one of the simplest Jazz3D programs there is!

```
import java.awt.*;
import java.applet.*;
import com.sygem.jazz3d3.*;

public class testSpheres
extends Applet implements Runnable {

   Thread myThread;
   World myWorld;
   int lightId;
   int objId;

   public testSpheres() {
   }

   public void init() {
       myWorld = new World(this);
       add(myWorld);
       RenderSolid gShader = new RenderSolid();
       Sphere3d sp1 = new Sphere3d(15,15,0,0,5);
       gShader.setDrawingMode(Render.GOURAUD);
       sp1.setRenderer(gShader);
       objId = myWorld.addObject(sp1);
       Light tempLight = new Light(0,0,1);
       lightId = myWorld.addLight(tempLight);
   }
```

```
    public void start() {
        if (myThread == null) {
            myThread = new Thread(this);
            myThread.start();
        }
    }

    public void stop() {
        if (myThread != null) {
            myThread.stop();
            myThread = null;
        }
    }

    public void run() {
        double x,y,z;
        x = 2;
        y = 3;
        z = -1;
        myWorld.prep();
        Object3d myObj;
        while (true) {
            myObj = myWorld.getObject(objId);
            myObj.rotateLocal(x,y,z);
            myWorld.redraw();
        }
    }
}
```


Exercise:

Try typing this program in. Change the amount the object is rotated by each time. Also try the different types of primitives available. This is just the start…

That's all folks!

But remember - your imagination is your only constraint!

Enjoy Jazz3D!