



Jazz3D Documentation

The easiest way to create truly interactive,
real-time 3D web content.

The logo for Jazz3D, featuring the text "Jazz3D" in a colorful, 3D-style font. The letters are red, yellow, blue, purple, red, and green respectively, with a 3D effect and a slight shadow.

Table of Contents

INTRODUCTION	4
WHY CREATE ANOTHER 3D API?	4
AIMS AND OBJECTIVES OF JAZZ3D.....	4
THE FUTURE	4
SHAREWARE	5
THE BASICS	6
COMPATIBILITY	6
DIMENSIONS	6
SPEED	6
BASICS OF 3D ENGINES	7
BRAVE NEW WORLD.....	8
THE 'WORLD' OBJECT	8
WHAT CAN I DO IN MY WORLD?	8
MOVING THE WORLD	9
WORLD ATTRIBUTES	9
<i>Bounding Boxes</i>	9
<i>Mouse Tracking</i>	10
<i>Backgrounds</i>	10
TEXTURES.....	11
<i>Animated Backgrounds</i>	11
PRODUCING THE FINAL IMAGE.....	12
<i>Resolution</i>	12
RENDERING SYSTEMS.....	13
FLEXIBILITY IS THE KEY	13
MORE DETAIL	14
<i>Wireframe (renderwf)</i>	14
<i>Flat Shaded (renderfs)</i>	14
<i>Gouraud Shaded (rendergs)</i>	15
<i>Texture Mapping (rendertm)</i>	16
<i>Perspective Corrected Texture Mapping (rendertmp)</i>	17
OBJECT D'ART.....	18
AVAILABLE PRIMITIVES.....	18
CREATE THE OBJECT	18
ADDING OBJECTS TO YOUR WORLD	18
OBJECT ATTRIBUTES	19
<i>Size</i>	19
<i>Colour</i>	19
<i>Ambient Colour</i>	20
<i>Visibility</i>	20
MOVING OBJECTS.....	20
<i>Translation</i>	20
<i>Rotation</i>	21
SPECIFIC PRIMITIVE INFORMATION	21
<i>Cube</i>	22
<i>Pyramid</i>	23
<i>Cylinder</i>	24
<i>Sphere</i>	25
<i>Torus</i>	26
<i>Checkerboard</i>	27
<i>Hemisphere</i>	28

LET THERE BE LIGHT!.....	29
TYPES OF LIGHT SOURCE	29
<i>Directional Light (light.class)</i>	29
<i>Point Light (lightpoint.class)</i>	29
<i>Spot Light (lightspot.class)</i>	29
CREATING A LIGHT.....	29
LIGHT PROPERTIES	30
<i>Spot Light</i>	30
ADDING THE LIGHT.....	31
LIGHT MOVEMENT	31
<i>Translation</i>	31
<i>Rotation</i>	31
LIMITATIONS.....	32
LESS PRIMITIVE PRIMITIVES	33
LOADER OBJECTS	33
LATHE OBJECT.....	34
FONTS.....	35
TEXT OBJECTS.....	35
MISCELLANEOUS INFORMATION.....	37
API INFORMATION.....	37
BENCHMARKS	37
<i>Benchmark 1</i>	38
<i>Benchmark 2</i>	38
<i>Benchmark 3</i>	38
<i>Benchmark 4</i>	38
API STRUCTURE.....	39
SAMPLE PROGRAM WALKTHROUGH.....	40
FIRST STEPS	40
GLOBAL VARIABLES.....	40
CREATING THE WORLD.....	41
THREADS	42
RUN-TIME OBJECT MANIPULATION	42
THE FINISHED ARTICLE.....	43

Chapter

1

Jazz3D Documentation

The easiest way to create truly interactive, real-time 3D web content.

Introduction

Why create another 3D API?

My reasons for creating a 3D API for Java were many-fold. First and foremost, there was the challenge. I had long been interested in 3D graphics, but never had a computer powerful enough, or a language as capable as Java. So when I began writing a small program to draw triangles on the screen, I soon realised that the potential was there to create a full blown API.

Another thing that spurred me 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. I wanted Jazz3D to be very easy to use, and I have designed it with this in mind.

It wasn't just the complexity of other products that made me create Jazz3D, however. Each available API had it's own set of unique features, and I wanted to create an API which encompassed the best of the rest - although I am currently some way from that, I hope that I will get closer to my goals in time.

Aims and Objectives of Jazz3D

I guess 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, 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.

Of course, any 3D API must rely on it's speed. However, when designing this product, I didn't want to sacrifice too much ease of use for that speed. I am sure that I could have optimised the code still further, but this would have lead to a considerably more cluttered API. Despite these design decisions, Jazz3D still stands up well against the competition in terms of speed. Further efforts will be made to optimise the core classes of the library, and these will be released as soon as they are available.

The Future

There are a great many potential improvements I would like to make to Jazz3D. Some are more important than others, and some of them are easier to implement than others. I will keep a list of future enhancements up-to-date on my web site.

For now, here is a taster of things to come. I have tried to order according to the likelihood that they will be implemented. Some of them seem quite tricky, so they may get pushed to a later release at some point.

- Convert to matrix math internally
- Create a Camera object to ease movement
- Image post-processing system
- Hierarchical object model
- More object loaders (to be determined)
- Animated textures
- Collision detection
- Shadows in real-time
- Transparent objects / faces
- Landscape objects
- Span-buffer renderer
- Extruded objects
- Phong shading renderer
- Environment mapping renderer
- Object morphing
- Sub-pixel / Sub-textel accuracy

Phew! Quite a list, isn't it! If all of that gets done, I'm sure you'll agree that Jazz3D will be one heck of a system.

Shareware

Yes, Jazz3D is a SHAREWARE product. This means that you can distribute the full product as you like, as long as the zip file remains intact. You can use it without restriction on your web pages. However, the Shareware version will print to the screen an 'annoying shareware message' - obscuring your final images and somewhat spoiling the effect. This can be removed - by paying the Shareware fee of \$25 (US).

There are 2 ways of doing this. If you are on the internet, you can pay using our on-line ordering system. Point your browser at <http://order.kagi.com/?V49&S>, and follow the simple instructions! Alternatively, you can use the 'register.exe' application, included with this package. This will allow you to use various methods of payment, including cash, VISA and Amex. Again, this is an easy to use application, so just follow the instructions and away you go!

Once we have received confirmation of your payment, we will send you (via email) an updated version of Jazz3D, which will not display this annoying message any more. I hope this provides enough of an incentive for you to pay up!

It may be of interest to you that this annoying message has a far greater impact upon Netscape users than it does for Internet Explorer - check out the difference if you can!

The Basics

Compatibility

2 versions of Jazz3D have been provided for your use - one is compatible with JDK 1.02, and one is to be used with JDK 1.1.2 or higher. At present, Jazz3D is not supported with Java 2 - it may work, but it hasn't been tested.

The only difference to you the user is in the initial 'import' statement you must use. For the JDK 1.02 version, use the following import statement;

```
import jazz3d_102.*;
```

And for JDK 1.1.2 or higher;

```
import jazz3d_112.*;
```

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). It also features the 4th dimension - without it we could not create displays which move!

Speed

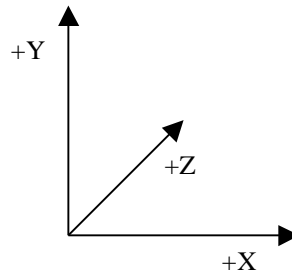
As I mentioned in the previous chapter, speed is very important to a 3D engine, and to you the customer as well. I have tried to make this initial release of Jazz3D as fast as I can - and as time goes by I will no doubt improve the speed still further. I want this to be the fastest thing there is in 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. These can give considerable performance gains. Another trick I have come across is avoiding the use of arrays wherever possible - these can be very slow, due to all the bounds-checking Java performs. I removed a large number of array accesses 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. 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 I 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 any of the points of the face are on-screen - if none of them are, we assume the face is off screen (this assumption can lead to rendering errors in rare cases, but the speed trade-off is worth it). 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 edge tables
- Draw the face
 - Start at the top - constant y value for each line
 - Draw the line 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 can add colour information, shading, textures - whatever we want. But that would be giving away too much, so read on for more detailed instruction on Jazz3D - the only choice for real-time 3D web content.

Brave 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 my_world = new world(this);
```

Wow! How easy is this! Let's hope the rest of the API can keep up (which of course, it can). Note that you need to pass an Applet as the parameter to the world object. This doesn't mean that Jazz3D can only be used in Applets, however. To make it work for applications, just write your program as an Applet, and add a main() method to it. There are plenty of examples of this in most Java references.

The next essential step is to add the world object to your applet or application. This is done in the same way as you would add any other AWT component. This is because the world class is implemented as an extension of the AWT Canvas class.

```
add(my_world);
```

Of course, this can be used in conjunction with any layout manager you choose. Note that if you resize the world object, only the size of the canvas will change. This means that the objects, will not be resized on screen, and you will just see a smaller section of the main display.

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.

```
my_world.prep();
```

What can I do in my world?

Well, given that this is meant to represent the real world in structure, a world contains several important things, like objects, lights and cameras. The vital step in creating your world comes when you add these objects to the world. Fortunately, it couldn't be simpler than with Jazz3D.

When you add something to the world, an integer is returned. It is useful to declare this globally, because you will need it later to manipulate your world's contents.

```
int object_id;  
object_id = my_world.addObject(obj, renderer)
```

Creating objects will be described in chapter 5, whilst the rendering system is described in the next chapter, (chapter 4).

Moving the world

The world class also allows you to move around the objects it contains. These are provided in addition to the object rotation and translation commands.

Let's say you wanted to move towards an object which is directly in front of you. An easier way to visualize this is to think of what the object is doing. In the real world, of course, the object wouldn't move - but in 3D space, it is moving towards you. So rather than move the viewpoint, we move all the objects instead. To move towards an object, we actually move that object (and all others) towards you. With that in mind, here are the 3 methods for moving around the world.

```
my_world.translateX(x);
```

```
my_world.translateY(y);
```

```
my_world.translateZ(z);
```

Each of these methods takes a double precision number as a parameter, which represents the distance you wish to move, in Jazz3D world units. All they really do is move all objects in the world the same distance, in the opposite direction.

A single method is provided for rotation of the world - this is all that is needed.

```
my_world.rotate(x,y,z);
```

This method takes 3 integer parameters - the number of degrees to rotate in each axis. Again, what this actually does is rotate all objects and light sources in the opposite direction.

World attributes

Bounding Boxes

The default visibility check for objects works on the level of faces - triangles of 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 complex scenes.

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 I realised this, I 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 one or more of them are on-screen, the object is deemed visible. The object is then rendered normally. If none of the vertices are on-screen, then the object cannot be seen, and so is not drawn.

As you can probably see, 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 of those faces, only the eight vertices will be checked! This allows much more complex scenes to be handled efficiently.

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

```
my_world.setBoundingBoxes(true);
```

And to turn off 'bounding box' mode;

```
my_world.setBoundingBoxes(false);
```

These commands can be issued at any time during the execution of your program.

Mouse Tracking

I believe that this is one of Jazz3D's most unique features, compared to the other 3D Java APIs available. If mouse tracking is enabled, 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 coordinate.

Activating mouse tracking is done a similar way to 'bounding boxes';

```
my_world.setMouseTracking(true);
```

And to deactivate mouse tracking;

```
my_world.setMouseTracking(false);
```

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;

```
my_world.getObjectAt(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. A detailed example of this is given in Appendix A, under the heading 'Mouse Tracking'.

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? OK then, take a look at this.

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

That would set the background of your applet to a lovely shade of red. I think you get the idea.

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.

```
my_world.setBackground("image.gif");
```

```
my_world.setTiledBackground("image2.jpg");
```

This image does not have to be static, however. The world object provides 4 methods to allow you to animate the background image. These methods are;

```
my_world.shiftBgLeft(int);  
  
my_world.shiftBgRight(int);  
  
my_world.shiftBgUp(int);  
  
my_world.shiftBgDown(int);
```

The integer being passed in to these methods controls the number of pixels the image will be shifted by. Note that this is in screen pixels, not texture pixels (texels). This is especially relevant if you are using a stretched background image.

You can also pass in a 'texture' object directly to the 'setBackground' and 'setTiledBackground' methods. This is useful if you are using the texture mapper and want to re-use your textures. It also makes it possible to have animated backgrounds. Read on for more details...

Textures

This is a good opportunity to introduce the idea of 'textures' - the way that Jazz3D deals with images internally. 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 your 'world' 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 = my_world.loadImage(filename);
```

'filename' must be a String representing the relative path to your image from the applet. Supported image formats are GIF and JPEG - this is not a limit imposed by me, but by the Java language. These are the only two image formats Java will load without the need for external software. In my experience, the JPEG loader in version 1.02 of the JDK is a little flakey - in fact, it regularly fails to load an image. However, I have never experienced any problems with the GIF loader. So, I would use GIFs if I were you.

This will load your image into Jazz3D, where you can do one of two things with it: set the background image, or load it into the texture mapper.

Animated Backgrounds

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

These textures are also 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.

Producing the final image

Inside of your main program loop (see Chapter 9 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. This is done with a simple method call;

```
my_world.redraw();
```

Inside this method, Jazz3D will produce the image, calling the relevant renderers, setting up the mouse-tracking buffer if necessary, 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.

```
my_world.setDelay(10);
```

The parameter is just an integer specifying the number of milliseconds Jazz3D should sleep for.

Resolution

The size of the final image is obviously dependant 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'.

Rendering Systems

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:

- Wireframe
- Flat shaded
- Gouraud shaded
- Gouraud shaded, affine texture mapping
- Gouraud shaded, perspective corrected texture mapping

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

```
(1) renderwf wireframe = new renderwf();  
(2) wireframe.setLineColour(255,0,0);  
(3) cube3d cube_obj = new cube3d(0,0,1);  
(4) int cube_id = my_world.addObject(cube_obj, wireframe);
```

That's all there is to it! In step 1, we create our renderer object. This could be any of the 5 currently implemented renderers. Then, in step 2, we perform any operations specific to each renderer. Here, for example, we call a method to specify the colour of the line drawing mode. These renderer specific methods are described in more detail shortly. Step 3 is where we create the object we want to add to the world. These are described in more detail in chapter 5. Finally, in step 4 we add the object to the world. This is where the renderer becomes linked to the object.

Note that this creates a dependency between the object and the renderer - if you change some aspect of the renderer (through one of the method calls detailed below), it will alter all instances of that renderer. So, if you want 2 wireframe objects, with 2 different colours, you would need 2 different instances of 'renderwf'.

More Detail...

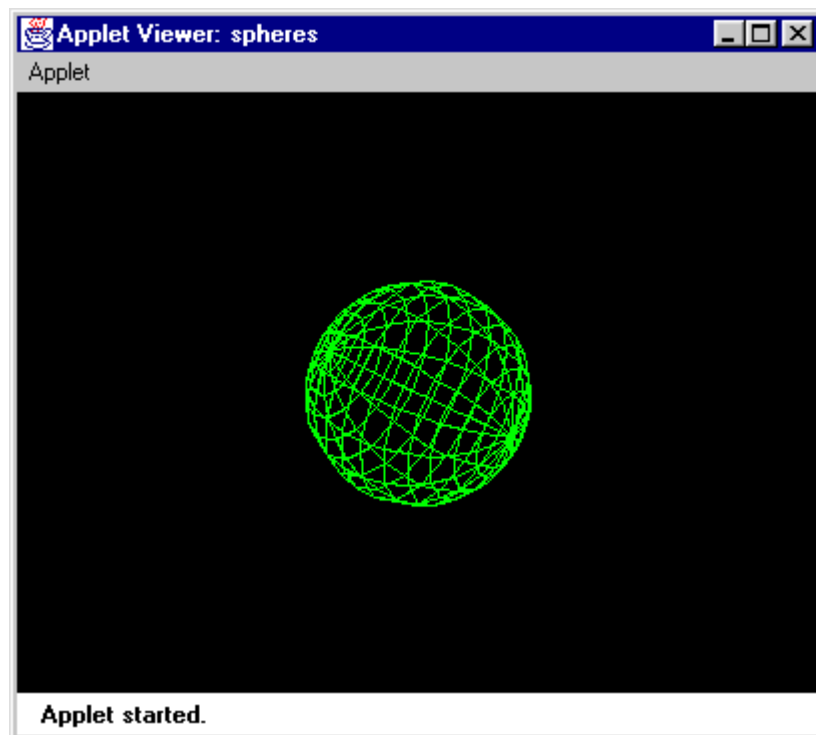
Wireframe (renderwf)

The wireframe mode is the simplest of all the rendering modes. It is not subject to any lighting, and it only draws the outline of each face. At present mouse tracking is not supported in this mode (the user would need to click on the lines between points, which is a bit tricky).

The only 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;

```
renderwf wireframe = new renderwf();  
wireframe.setLineColour(0,255,0);
```

This would set the line colour to a bright green.

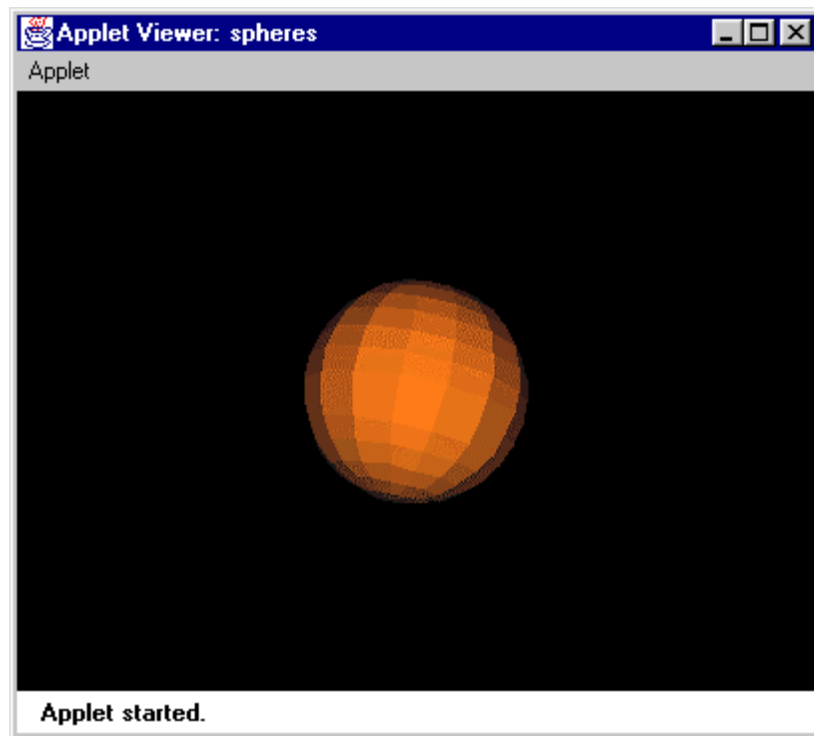


Note: the default line colour is white - (255,255,255). Also, any objects using the wireframe renderer do not require a lightsource - no shading is performed.

Flat Shaded (renderfs)

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.

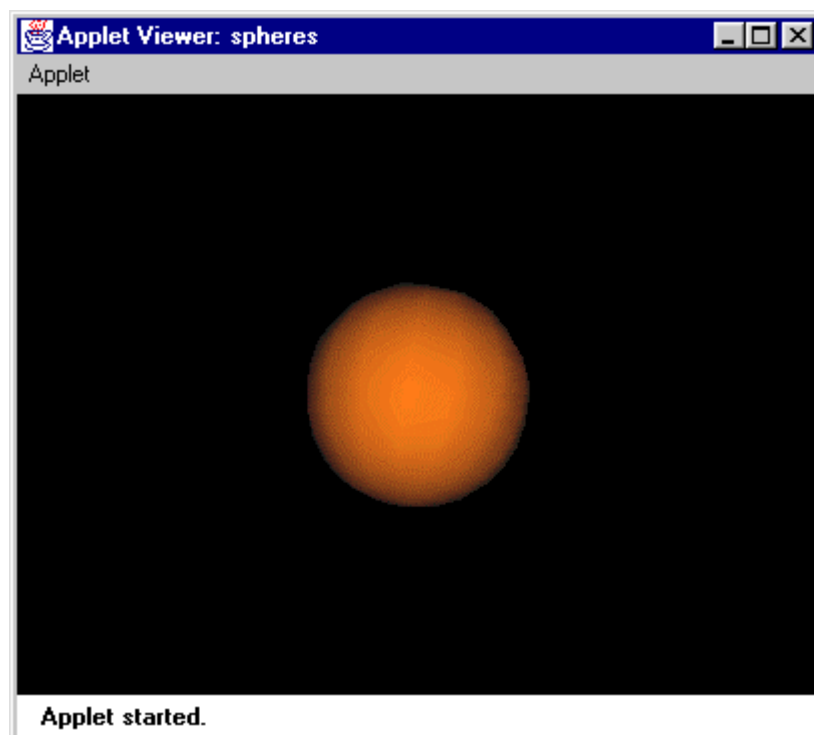
The lighting calculation for flat shaded mode takes the light intensity at the center point on the face and assumes this value is the same for each pixel on that face. This is then combined with the colour of the face to produce a colour value, again the same over the whole face. There are no specific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



Gouraud Shaded (rendergs)

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 average colour of all the faces which use that vertex. It then interpolates these values - first across each edge (scan-conversion), then across each scanline (drawing).

There are no specific method calls for the gouraud shaded renderer.



Texture Mapping (rendertm)

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 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, and I have tried to make it as fast as possible.

The shading in the texture mapper is of the gouraud flavour, rather than being flat shaded. For texture mapping, I felt that the extra image quality was worth the slight speed penalty. Jazz3D may, at some point in the future, offer a choice of shading modes.

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

```
tm.setImage(tex);
```

Only one texture is supported per object at present. This is another item on the long list of improvements to Jazz3D.

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

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.

```
tm.WrapUV(true);
```

The wrapping can be turned off by specifying 'false' rather than true.

The following picture gives a good example of the kind of quality images you can achieve with the texture mapper - this is a sphere, with 10 by 10 subdivisions, and an image of the earth wrapped into it. Only one, simple light source is used. Not bad, eh!



Perspective Corrected Texture Mapping (rendertmp)

This renderer works in exactly the same way as the previous texture mapper. The only difference is that it produces slightly better results, and is a little slower.

This image is taken from the same applet as the one above, only using the perspective corrected texture mapper. Although the differences are small, the overall effect is better.



Object D'Art

Available primitives

This is the list of primitives currently implemented in Jazz3D.

- Cube
- Pyramid
- Cylinder
- Sphere
- Torus (donut)
- 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 new_object = new cube3d(0,0,8);
```

This will create a cube 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 final classes. It also forces the user (that's you) to follow the same sequence for every object - create, resize, re-colour, add to world. I believe that this makes the programs written using Jazz3D easier to read - it is obvious what colour an object should be, where it should be and how big it should be.

Adding objects to your world

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

```
int obj_id = my_world.addObject(new_object, renderer);
```

It really is as simple as that. The integer which is returned can be used later on to access the object through the world. This is the only way that you can access objects at run-time, so it is important that you keep track of this variable if you need it.

Object Attributes

Before you add your object to the world, you may decide to override the default attributes for the object. Currently there are 4 major attributes you can set - these methods work in the same way for every Jazz3D object, maintaining a consistent interface throughout the API.

Size

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

```
new_object.scaleObject(double, double, double);
```

The three double precision numbers correspond to the desired size in the x, y and z dimensions. Note that if you have bounding box mode turned on (see chapter 3), then the bounding box is automagically re-sized along with the object.

An interface to this method is also provided through the 'world' object, so you can modify your objects during run-time (although this can be a slightly time-consuming exercise, depending on the number of vertices your object contains).

```
my_world.scaleObject(obj_id, double, double, double);
```

Where 'obj_id' is the integer returned when you added the object to the world - this was described on the previous page, in case you have forgotten it.

Colour

All primitive objects are created with their colour set to white. Not very exciting, I know, so Jazz3D provides a way for you to change the colour of your objects, using the 'setColour' method call. Here is an example.

```
new_object.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 what about parts of an object? Not a problem. You can set the colour of a particular face like so:

```
new_object.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 of each face cannot be guaranteed, or even calculated on the fly. You'll just have to experiment if you want to use this method.

As with most of these methods, an interface is provided through the 'world' object. Just specify the id of the object you wish to alter. The method names are the same.

```
my_world.setColour(obj_id,255,0,0);
```

```
my_world.setFaceColour(obj_id,facenum,255,0,0);
```

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.

```
obj.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.

The ambient colour of an object can also be modified whilst the program is running - once again, through the world class. This allows you to modify the ambient colour during run-time (and if you can think of a use for that, let me know, because I can't!) Anyway, here's an example of how to do just that.

```
my_world.setAmbientColour(obj_id, 60, 0, 0);
```

This would set the ambient colour of the object to a dark-ish red. Lovely.

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;

```
obj.setVisible(true);
```

or

```
obj.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.

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? There are two basic forms of movement in the Jazz3D world - translation and rotation.

Translation

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

```
obj.translateX(0.5);
```

```
obj.translateY(-2.75);
```

```
obj.translateZ(43.245);
```

Of course, the numbers are just examples. As with many of the object3d methods, you will need to activate them through the world3d class once your program is running. The following methods allow you to do this.

```
my_world.translateObjectX(obj_id, 0.5);  
  
my_world.translateObjectY(obj_id, -2.73);  
  
my_world.translateObjectZ(obj_id, 3);
```

Rotation

There are two ways in which a Jazz3D object can be rotated. Firstly it can be rotated about it's own center point (which is the point you use when you define the object). This achieved using the following method call.

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

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 I included the ability to rotate an object around any given point. The method works like this;

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

Much the same as the previous version of the rotateXYZ 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.

Just like in object translation, rotation methods are provided in the world object, for use whilst your program is running. The format of these is very similar to the methods just described.

```
my_world.rotateObjectXYZ(obj_id,x,y,z);  
  
my_world.rotateObjectXYZ(obj_id,x,y,z,0,0,8);
```

The only difference between the 2 sets of methods is a slight change in the name, and the addition of an object specifier in the world methods. In addition to these, the world class contains 2 methods to allow all objects in the world to be rotated at the same time, around either their center points or a given point (again, point (0,0,8) is used in the example).

```
my_world.rotateAllObjectsXYZ(x,y,z);  
  
my_world.rotateAllObjectsXYZ(x,y,z,0,0,8);
```

Specific primitive information

This section contains anything I think you might need to know about each primitive - from how to create them, what their limitations are, and how to use them with each type of renderer. I have also included sample images of each primitive, generated using the flat-shaded renderer, and a single light source.

Cube

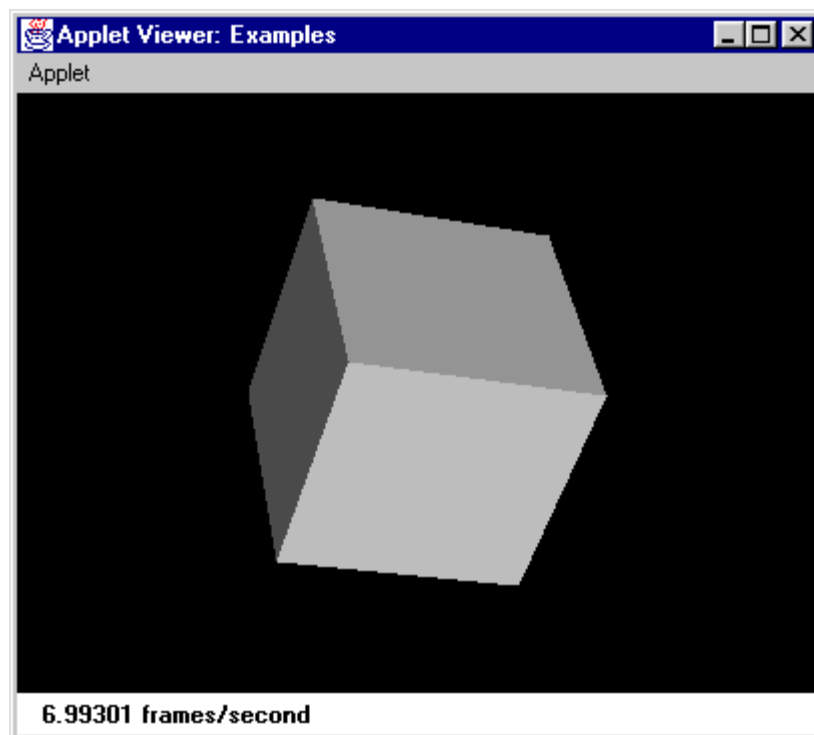
Constructor

```
cube3d(double x, double y, double z);
```

Details

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.

At present, the only type of texture mapping that the cube supports is the same texture on each face. Soon, this will change, and you will be able to specify the texture to use on each face. Note that UV wrapping is not supported on the cube object.



Pyramid

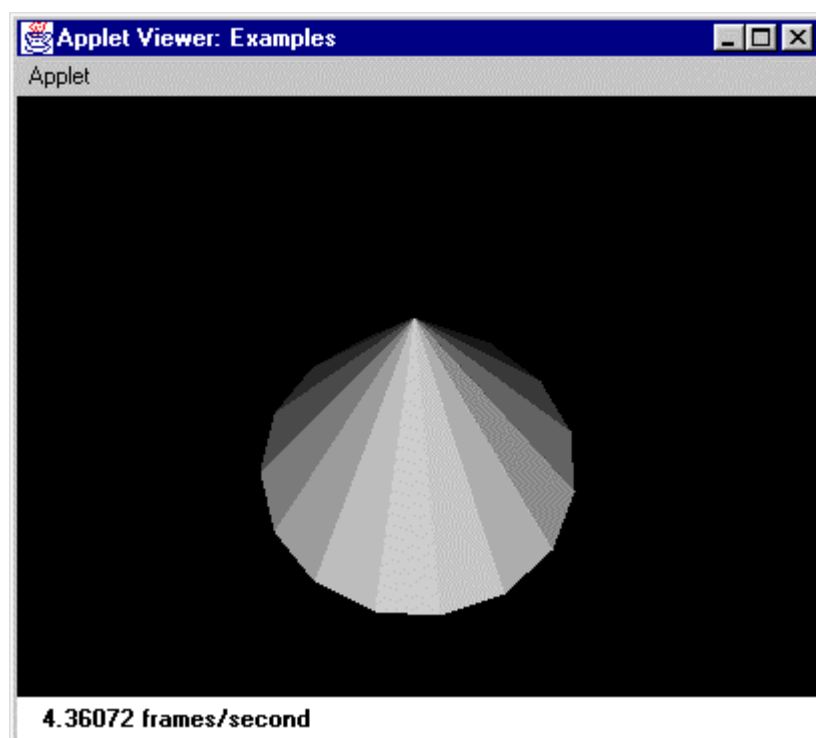
Constructor

```
pyramid3d(int sub, double x, double y, double z);
```

Details

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.

UV wrapping of textures is now supported for this primitive. Details of this are in chapter 4, under 'Texture Mapper'. Only one texture can be applied to the object at any time.



Cylinder

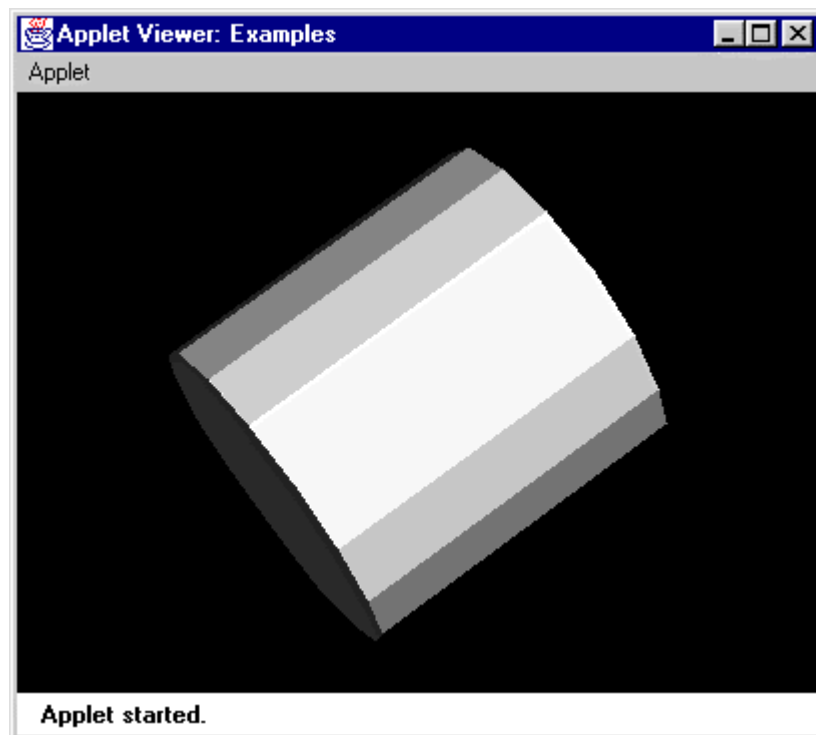
Constructor

```
cylinder3d(int sub, double x, double y, double z);
```

Details

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.

UV wrapping of textures is now supported for this primitive. Details of this are found in Chapter 4. Only one texture can be applied to the object at any time.



Sphere

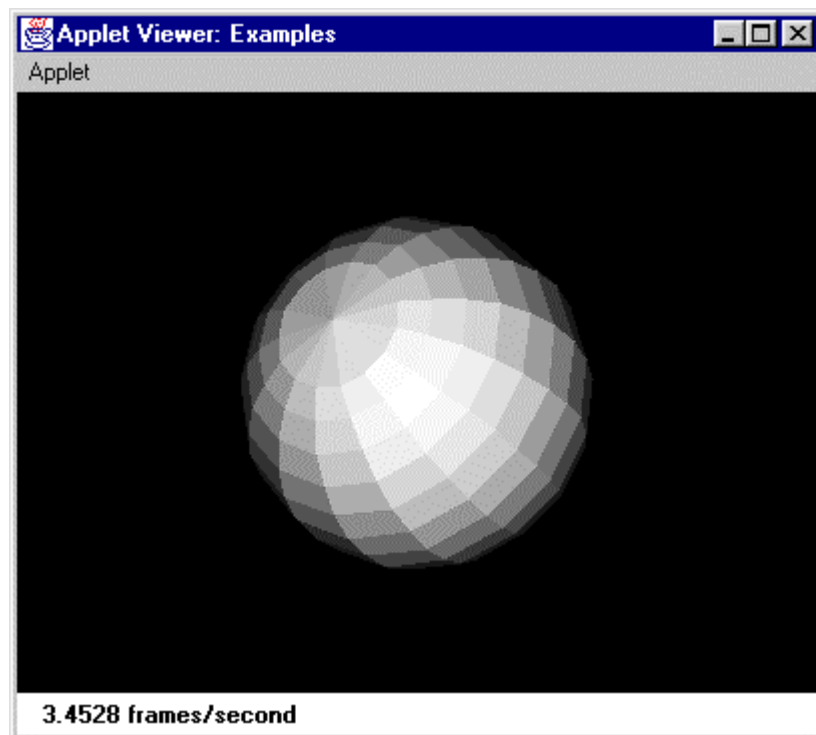
Constructor

```
sphere3d(int h,int w, double x, double y, double z);
```

Details

The two parameters 'h' and 'w' specify the number of vertical and horizontal subsections 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.

UV texture mapping is supported for the sphere object, but only one texture can currently be applied to the object. Details of this are in Chapter 4.



Torus

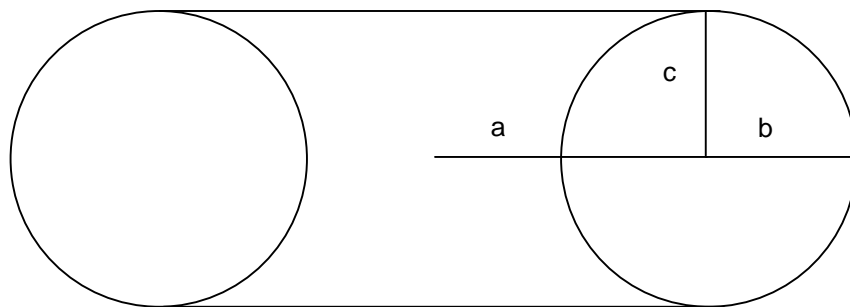
Constructor

```
torus3d(double a,double b,double c,int w,int h,double x,double y,double z);
```

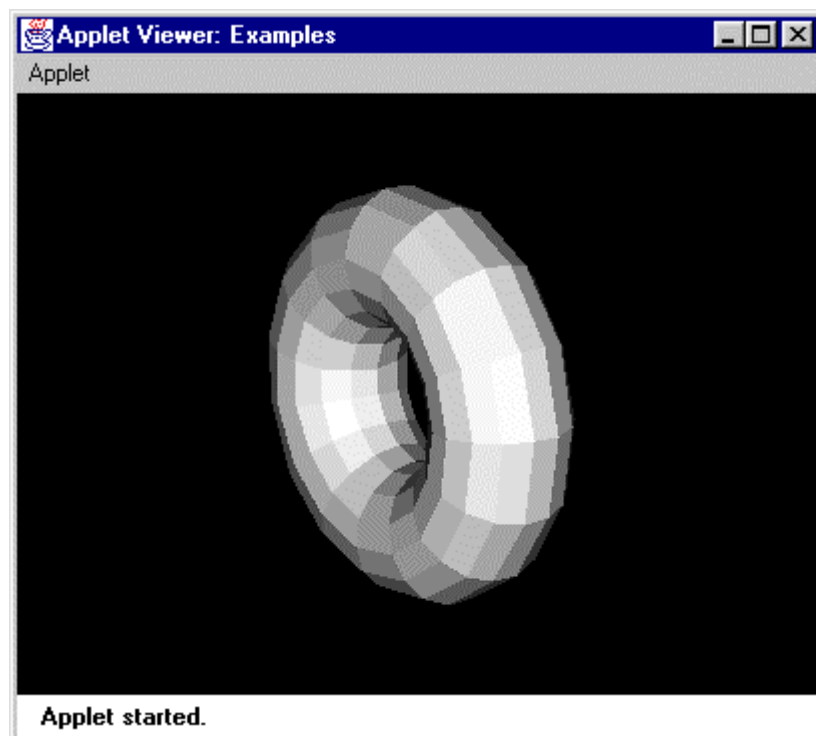
Details

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. The UV texture mapping is also supported for the torus object. See Chapter 4 for more details on this subject.



Checkerboard

Constructor

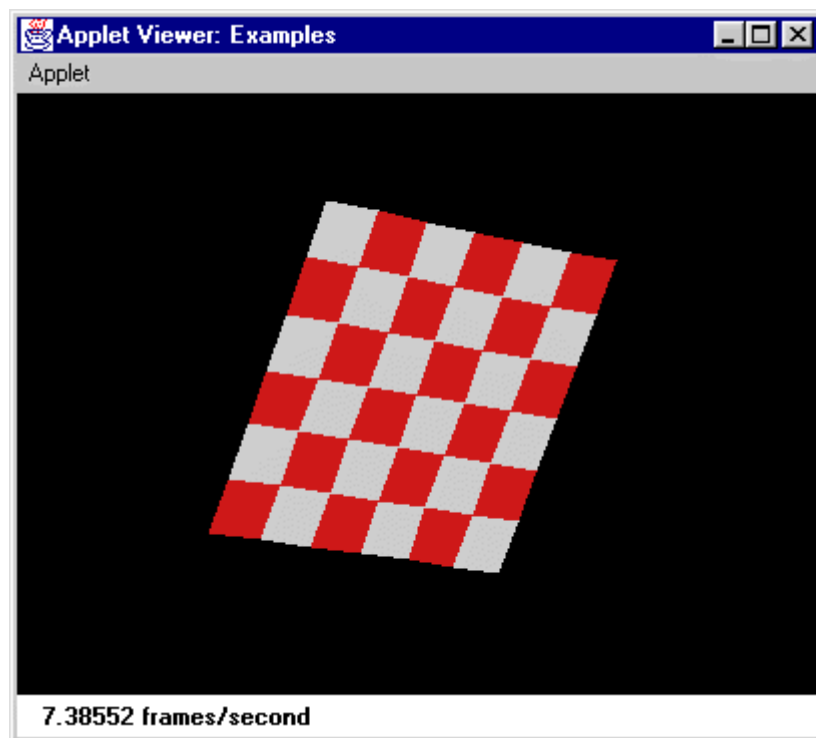
```
checkerboard3d(int a,int b,double x,double y,double z);
```

Details

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

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.

As a result of the gouraud shader's way of calculating the colour at a vertex, this primitive doesn't work well with the gouraud shader or texture mapper. It works best with the flat shader, although this may change in time.



Hemisphere

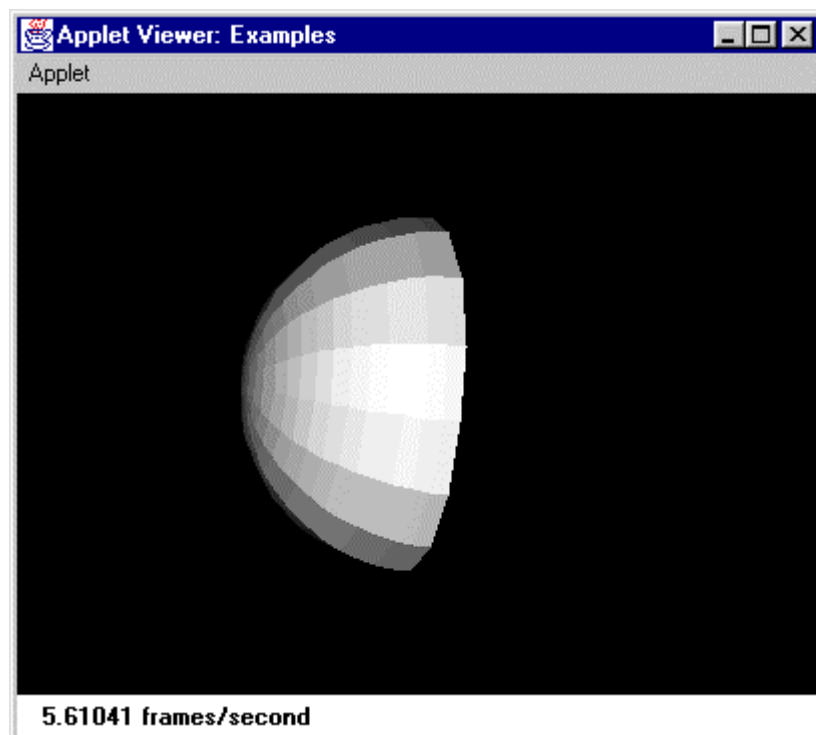
Constructor

```
hemisphere3d(int h,int w,double x,double y,double z);
```

Details

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.

UV texture mapping is supported for the hemisphere object. See Chapter 4 for more details on this subject.



Let there be Light!

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.class)

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 doesn't go down 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.class)

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.class)

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 5%.

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.

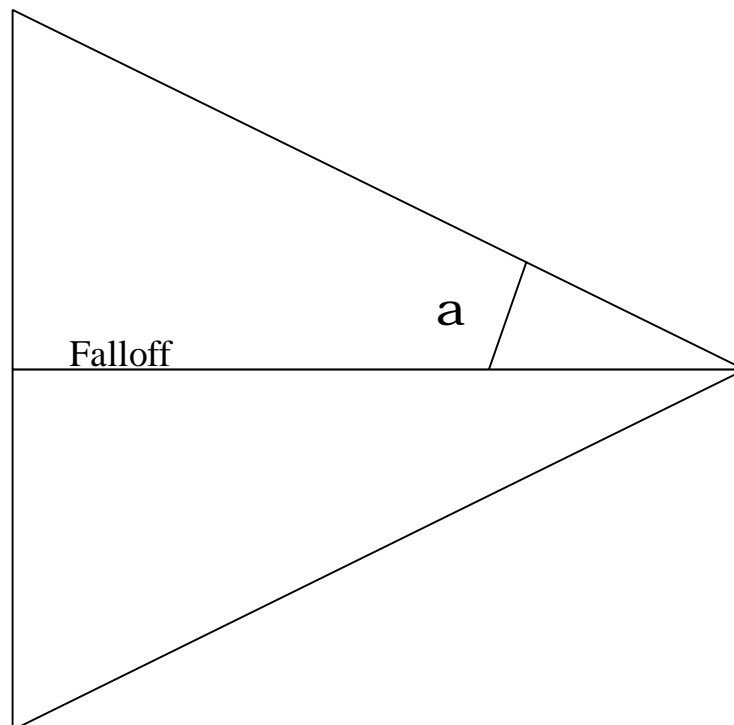
```
light1.setIntensity(double inten);
```

```
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.

```
light_id = my_world.addLight(light1);
```

The integer variable `light_id` can then be used to reference this light source through the world object. Again, this is just how objects work.

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;

```
light1.translate(double x, double y, double z);
```

Yep - that's it. And once the light has been added to the world, you should use the following method, from the `world3d` class. This uses the

```
my_world.translateLight(int id,double x,double y,double z);
```

These methods will translate your light source by the appropriate amounts in each direction. You will need to use the integer variable returned when you added the light to your world.

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;

```
rotateXYZ(int x,int y,int z);
```

```
rotateXYZ(int x,int y,int z,double px,double py,double pz);
```

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 single 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 its position around the origin. Rotating the light around a different point will also change the position of the light, just like for an object.

Spot lights become different again... When you rotate the light using the first rotate method, its position will also rotate around the origin. However, the point which the light is pointing at will not change. This must be done using the `pointAt()` method. The same is true for the second rotate method.

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.

Less primitive primitives

Loader Objects

Yes, Jazz3D CAN load objects created with 3D packages. 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)

The first step to loading an object file is to create an instance of a 'loader'. This class will hold the loaders for all of the supported formats you choose to use. Note that you only need the classes for the file types you want. The loader system is fully decoupled and extensible.

```
loader lo = new loader(my_world);
```

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.

```
lo.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'?

```
lo.assignLoader("GEO", new loadergeo());
```

As you can probably guess, once the file loader has been assigned, any number of objects of that type can be loaded. 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 its parameters. All we have to do next is call the 'loadModel()' method, passing in the filename and the loader object we just created.

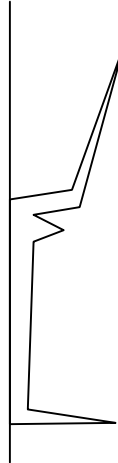
```
dragon.loadModel("Dragon.geo", lo);
```

Providing the model file can be found, the object is loaded and you can add it to the world in the normal fashion. At the moment, texture wrapping is not supported for any models. Also, the gouraud shader and texture mappers don't work well with 3DS files.

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.

```
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 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 new_obj = 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.

Fonts

This is included as something of a bonus feature in this initial release of Jazz3D - the ability to create 3D text in your world, which can be rotated and moved about just like any other object.

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 are stored in a directory call 'fonts' (no surprises there!) in the same location as the main Jazz3D class files. Inside this there is one directory per font, and in these subdirectories are the actual font files. Please note that only one font is included in the initial release - Arial.

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

```
font3d arial = new font3d("Arial", my_world);
```

This creates the font3d object and loads in the font file. The string for specifying the font name is case-sensitive, although as there is only one font at the moment this is less of an issue. The second parameter needs to be the world object you should have already created.

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. The Arial font included in this release contains the following characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

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 text = new text3d("Jazz3D",arial,0.5,0.01,0,0,8);
```

So, the first parameter is the actual text the object is to contain. This is just a String variable - nothing special here. At present, lowercase characters are not supported by the text3d object, and will be converted to uppercase - I'm working on this! Also, the only other characters which are supported are numbers. No punctuation or anything like that. Again, these will all appear in time.

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).

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 can then add it to your world in the same way as for the primitives. Here it is again, just for fun.

```
text_id = my_world.addObject(text, flat_shader);
```

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.

Chapter

8

Miscellaneous Information

API information

Number of Classes	34
Lines of Java code	4985
Internal Display type	Full Z-buffer, 24-bit display
Resolution	Any - determined by the user
Supported face shapes	Triangle and Quadrilateral
Rendering system	Applied on a per-object basis
Rendering types	Wireframe, Flat shaded, Gouraud shaded, Affine texture mapping, Perspective corrected texture mapping.
Lighting system	No limit on number of light sources
Lighting types	Directional lightsource, Point lightsource, Spotlight
Object loader system	Supports .GEO, .GEM, .ASC and .3DS files Fully extensible

Benchmarks

The following configurations were used for the benchmarking.

System 1

- Intel Celeron 333Mhz (Over-clocked to 400Mhz)
- 32Mb RAM
- Windows 98

System 2

- Intel Pentium II 333Mhz
- 96Mb RAM
- Windows NT 4.0

The applets used were running at a resolution of 400 x 300 - which I consider a little higher than for normal use. This resolution could put a considerable strain on less well endowed systems - and you don't want to irritate your viewers, do you? Note that 2 sets of figures are given - these represent the 2 versions of Jazz3D (the first one for JDK 1.02, the second is for JDK 1.1.2) to give you some idea of their relative speeds.

Benchmark 1

- 1 wireframe sphere
- No light sources
- Sphere rotating around its center point.
- See benchmark1.java

Max Frames per Second	Microsoft IE 5	Netscape Navigator 4.08
System 1	10.85 / 11.39	9.57 / 9.21
System 2	15.85 / 16.64	13.99 / 13.46

Benchmark 2

- 2 flat-shaded spheres
- 1 directional light
- Both spheres rotating around the same point.
- See benchmark2.java

Max Frames per Second	Microsoft IE 5	Netscape Navigator 4.08
System 1	9.90 / 11.12	7.93 / 7.61
System 2	14.39 / 16.16	11.53 / 11.07

Benchmark 3

- 2 gouraud-shaded spheres
- 1 point light
- Both spheres rotating around the same point.
- See benchmark3.java

Max Frames per Second	Microsoft IE 5	Netscape Navigator 4.08
System 1	9.29 / 9.39	7.41 / 6.96
System 2	13.03 / 13.17	10.40 / 9.77

Benchmark 4

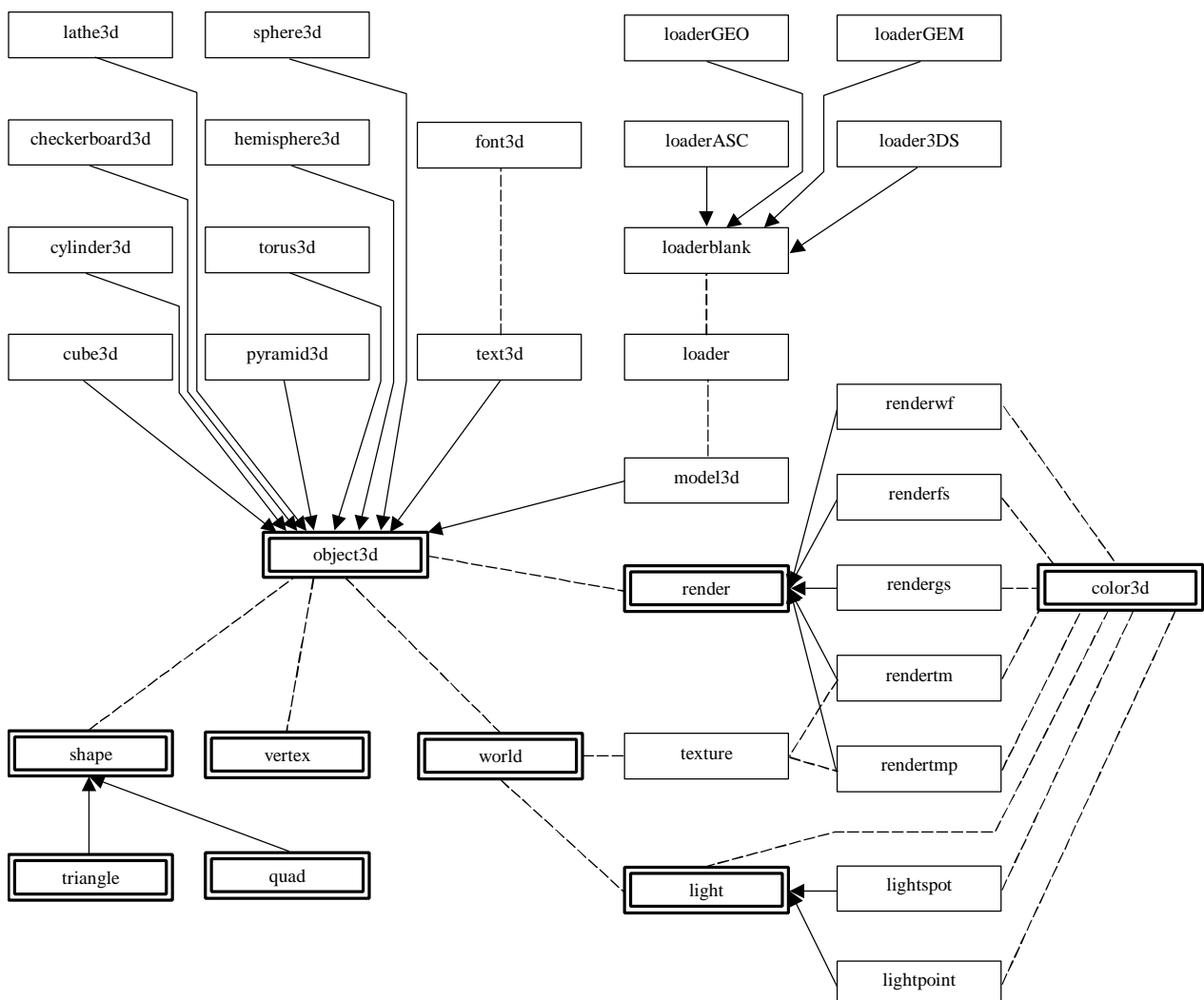
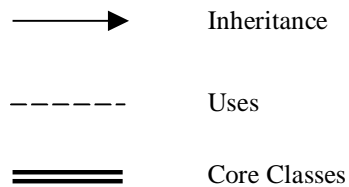
- 2 perspective corrected texture mapped spheres
- 1 spot light
- Spheres rotating around different points.
- See benchmark4.java

Max Frames per Second	Microsoft IE 5	Netscape Navigator 4.08
System 1	6.20 / 6.48	5.61 / 5.63
System 2	6.70 / 7.00	6.06 / 6.08

API Structure

This diagram is meant to represent the way in which objects in the Jazz3D API are linked together. It also highlights the core classes which are required in every Jazz3D program.

Here is a key to the diagram:



And if you can make anything out of that diagram, you're doing better than me! I just hope I don't add too many classes in the next version...

Sample Program Walkthrough

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 (for JDK version 1.02) - this is done using the import command, at the very top of the program, like this;

```
import jazz3d_102.*;
```

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 test_sphere extends Applet implements Runnable {
    public test_sphere() {
    }
}
```

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	my_thread	The thread object is required to get the program running - it also provides us with the ability to
world	my_world	This is the Jazz3D world - the focal point of the program
int	obj_id	An integer variable which will be used to reference the sphere we will add to the world
int	light_id	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 its constructor is an Applet - since we are already in the applet's `init()` method, we can refer to it as `this`.

```
my_world = 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(my_world);
```

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;

```
rendergs gouraud_shader = new rendergs();
```

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);
```

And now, we must add the sphere to the world. This associates the renderer with the object, and gives us the value we can use later to refer to the sphere through the world.

```
obj_id = my_world.addObject(sp1, gourard_shader);
```

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 temp_light = 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.

```
light1 = my_world.addLight(temp_light);
```

And that's about it! At the end of all that, our `init()` method should look like this;

```
public void init() {
    my_world = new world(this);
    add(my_world);
    rendergs gouraud_shader = new rendergs();
    sphere3d spl = new sphere3d(15,15,0,0,5);
    pid = my_world.addObject(spl, gouraud_shader);
    light temp_light;
    temp_light = new light(0,0,1);
    light1 = my_world.addLight(temp_light);
}
```

Threads

Remember that global variable called `my_thread`? 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 (my_thread == null) {
        my_thread = new Thread(this);
        my_thread.start();
    }
}

public void stop() {
    if (my_thread != null) {
        my_thread.stop();
        my_thread = 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.

```
my_world.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 integers, 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 `obj_id` variable as a pointer to the sphere.

```
z = -1;
```

```
y = 3;
x = 2;
my_world.rotateObjectXYZ(obj_id,x,y,z);
```

The final step is to force the world object to update it's display. This done with the following method call;

```
my_world.redraw();
```

And so the `run()` method will look like this;

```
public void run() {
    int x,y,z;
    x = 2;
    y = 3;
    z = -1;
    my_world.prep();
    while (true) {
        my_world.rotateObjectXYZ(obj_id,x,y,z);
        my_world.redraw();
    }
}
```

The finished article

Here it is - one of the simplest Jazz3D programs there is!

```
import java.awt.*;
import java.applet.*;
import jazz3d_102.*;

public class spheres extends Applet implements Runnable {
    Thread my_thread;
    world my_world;
    int light1;
    int obj_id;

    public spheres() {
    }

    public void init() {

        my_world = new world(this);
        add(my_world);

        rendergs gouraud_shader = new rendergs();

        sphere3d sp1 = new sphere3d(15,15,0,0,5);

        pid = my_world.addObject(sp1, gourard_shader);

        light temp_light = new light(0,0,1);
        light1 = my_world.addLight(temp_light);

    }

    public void start() {
        if (my_thread == null) {
            my_thread = new Thread(this);
            my_thread.start();
        }
    }
}
```

```
public void stop() {
    if (my_thread != null) {
        my_thread.stop();
        my_thread = null;
    }
}

public void run() {
    int x,y,z;
    x = 2;
    y = 3;
    z = -1;
    my_world.prep();
    while (true) {
        my_world.rotateObjectXYZ(pid,x,y,z);
        my_world.redraw();
    }
}
```