Textures

Need for Textures

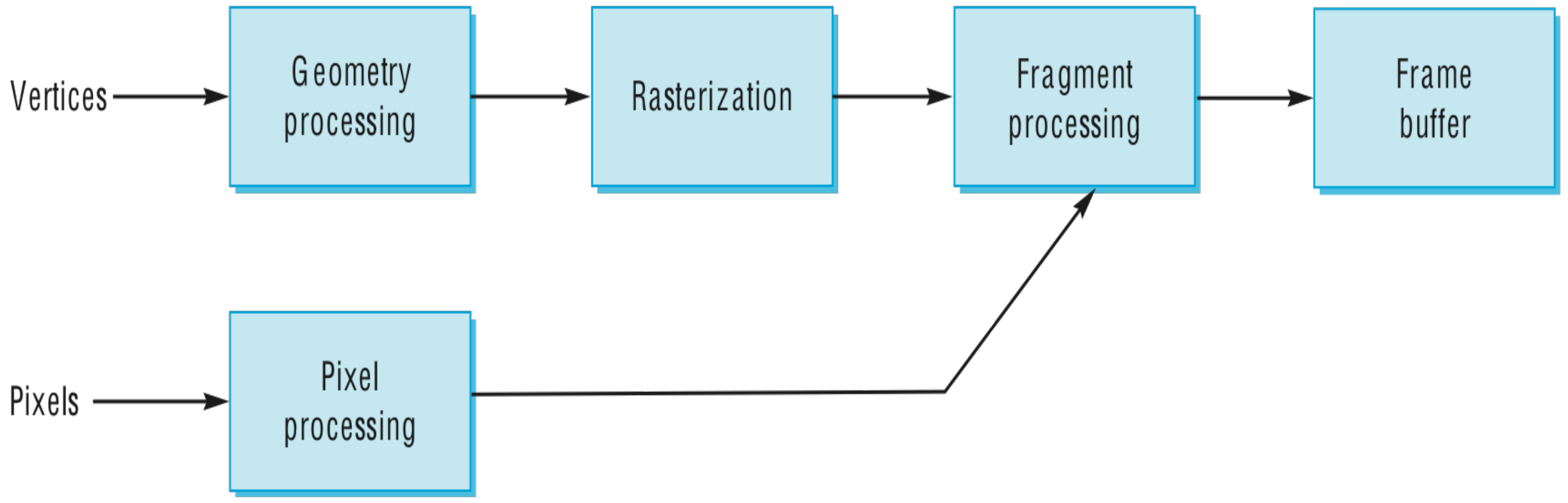
- Although graphics cards can render millions of polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin
- A source for these phenomena could be image data
- Image data generally has a one-to-one correspondence between a pixel in an image and a pixel on the screen
- When we apply image data to a geometric primitive, we call this a texture or texture map

Example

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
    - Too simple
- Replace sphere with a more complex shape
    - Does not capture surface characteristics (small dimples)
    - Takes too many polygons to model all the dimples
- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
- This process is known as texture mapping - uses images to fill inside of polygons

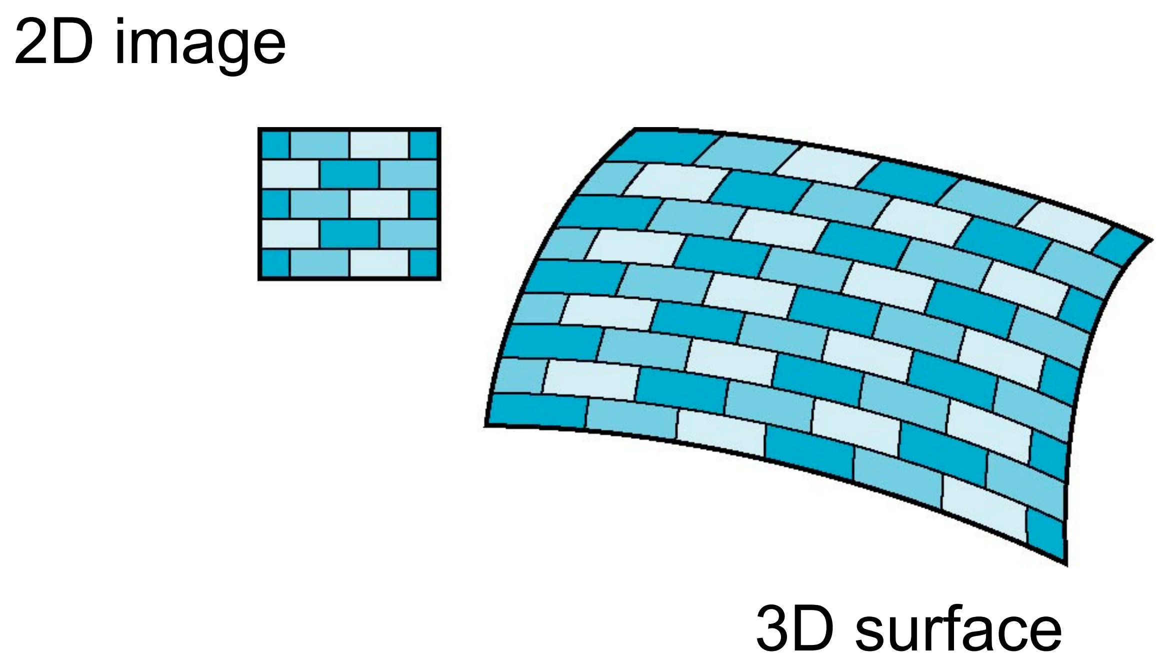
Pipeline

- Mapping techniques are implemented at the end of the rendering pipeline

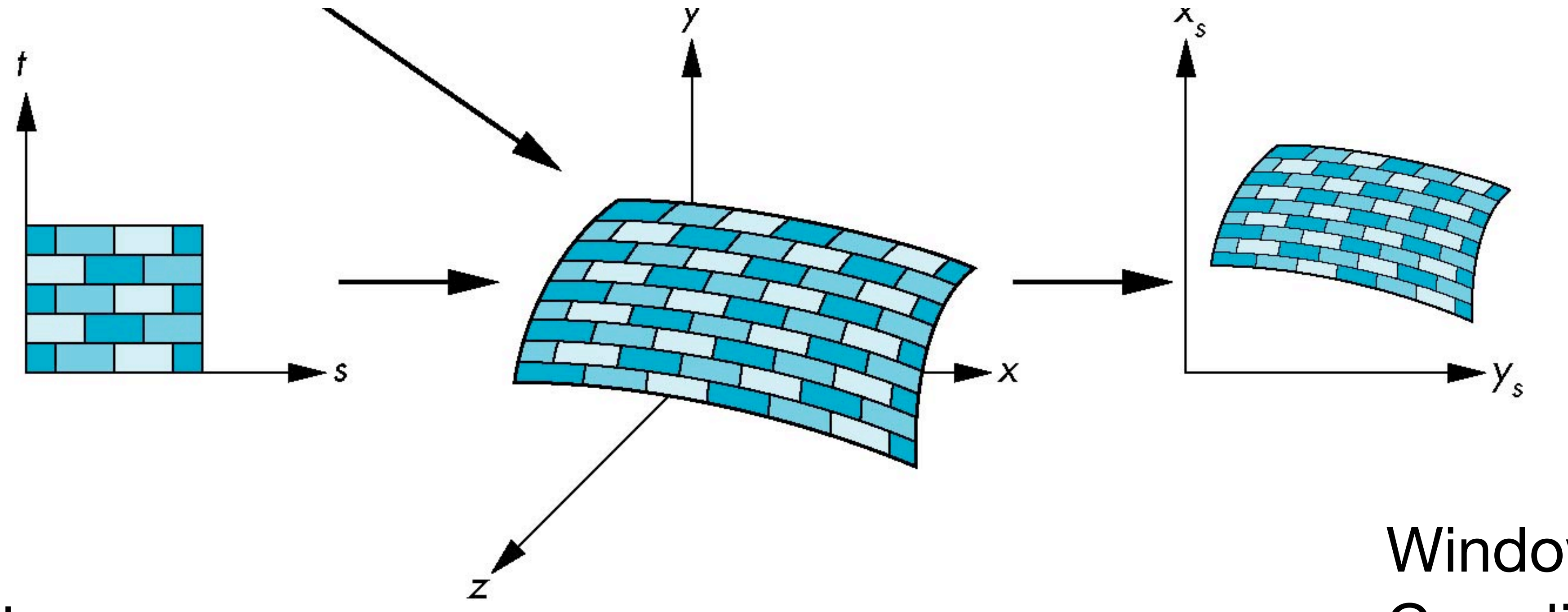


Mapping

- Although the idea is simple---map an image to a surface---there several coordinate systems involved



At Least 3 Co-ordinate systems



Texture coordinate - used to identify points in the image to be mapped

Object or World Coordinates - conceptually, where the mapping takes place

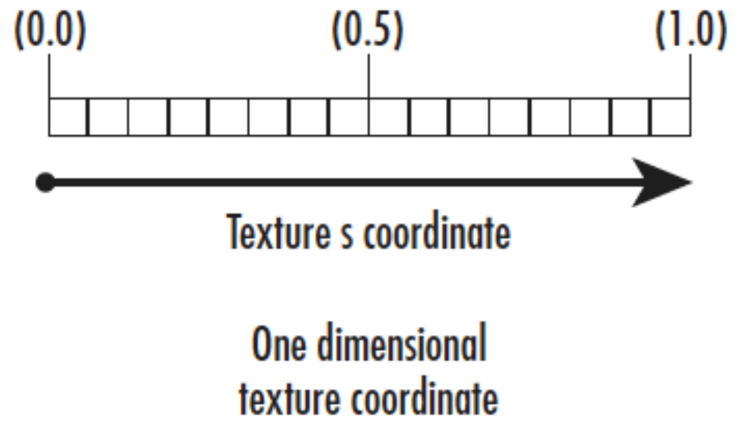
Window Coordinates - where the final image is really produced

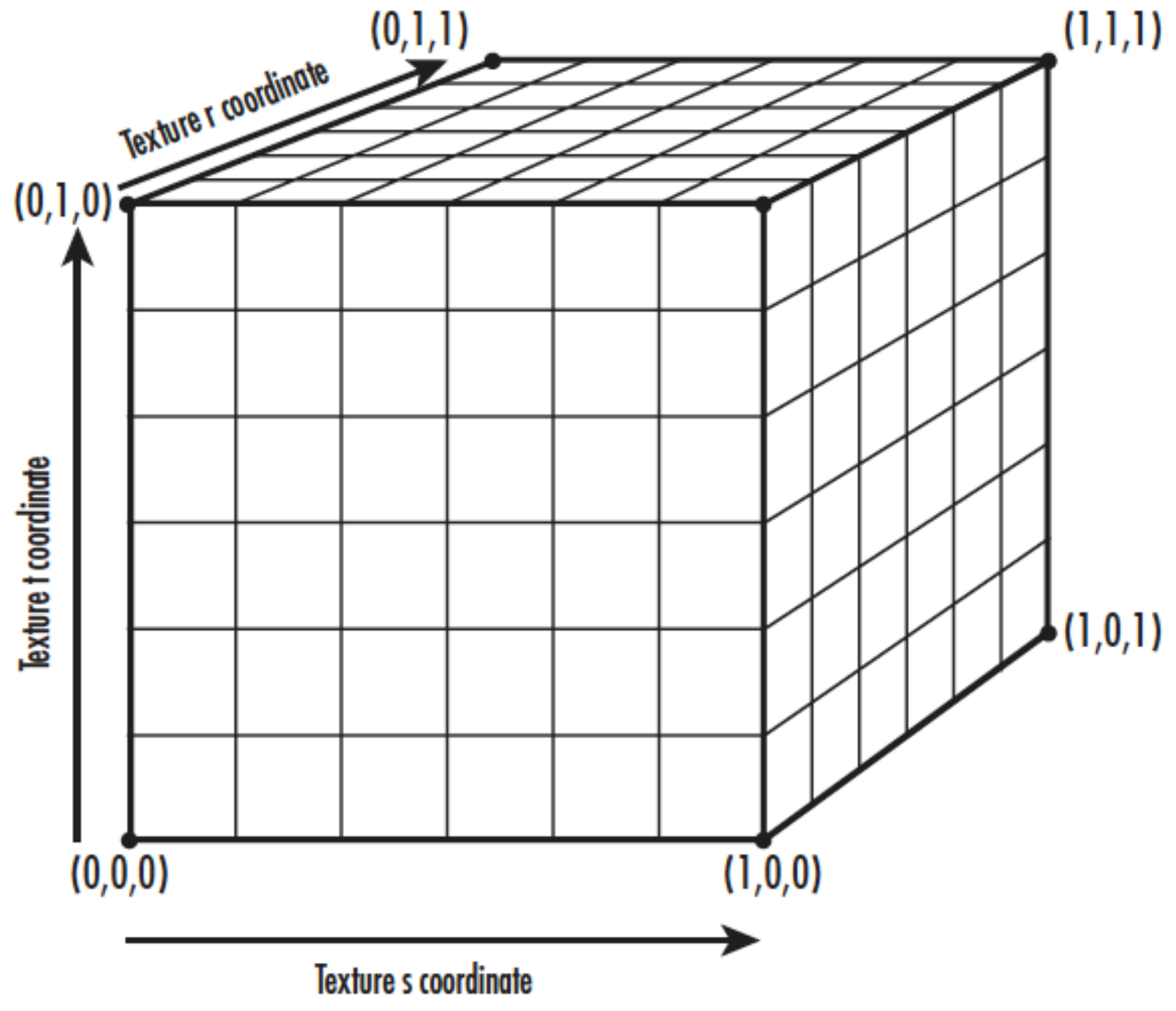
Mapping Textures to Geometry

- Loading a texture and enabling texturing cause OpenGL to apply the texture to any of the OpenGL primitives.
- You must, however, provide OpenGL with information about how to map the texture to the geometry. You do this by specifying a texture coordinate for each vertex - *Textels*
- Texels in a texture map are addressed not as a memory location (as you would for pixmaps), but as a more abstract (usually floating-point values) - texture coordinate.

Texture Coordinates - Textels

- Typically, texture coordinates are specified as floating-point values that are in the range 0.0 to 1.0.
- Texture coordinates are named s, t, r, and q (similar to vertex coordinates x, y, z, and w), supporting from one- to three-dimensional texture coordinates, and optionally a way to scale the coordinates.





void glTexCoord2f(Glfloat s, GLfloat t);

- You specify a texture coordinate using the glTexCoord function. Much like vertex coordinates, surface normals, and color values.
- One texture coordinate is applied using these functions for each vertex. OpenGL then stretches or shrinks the texture as necessary to apply the texture to the geometry as mapped.

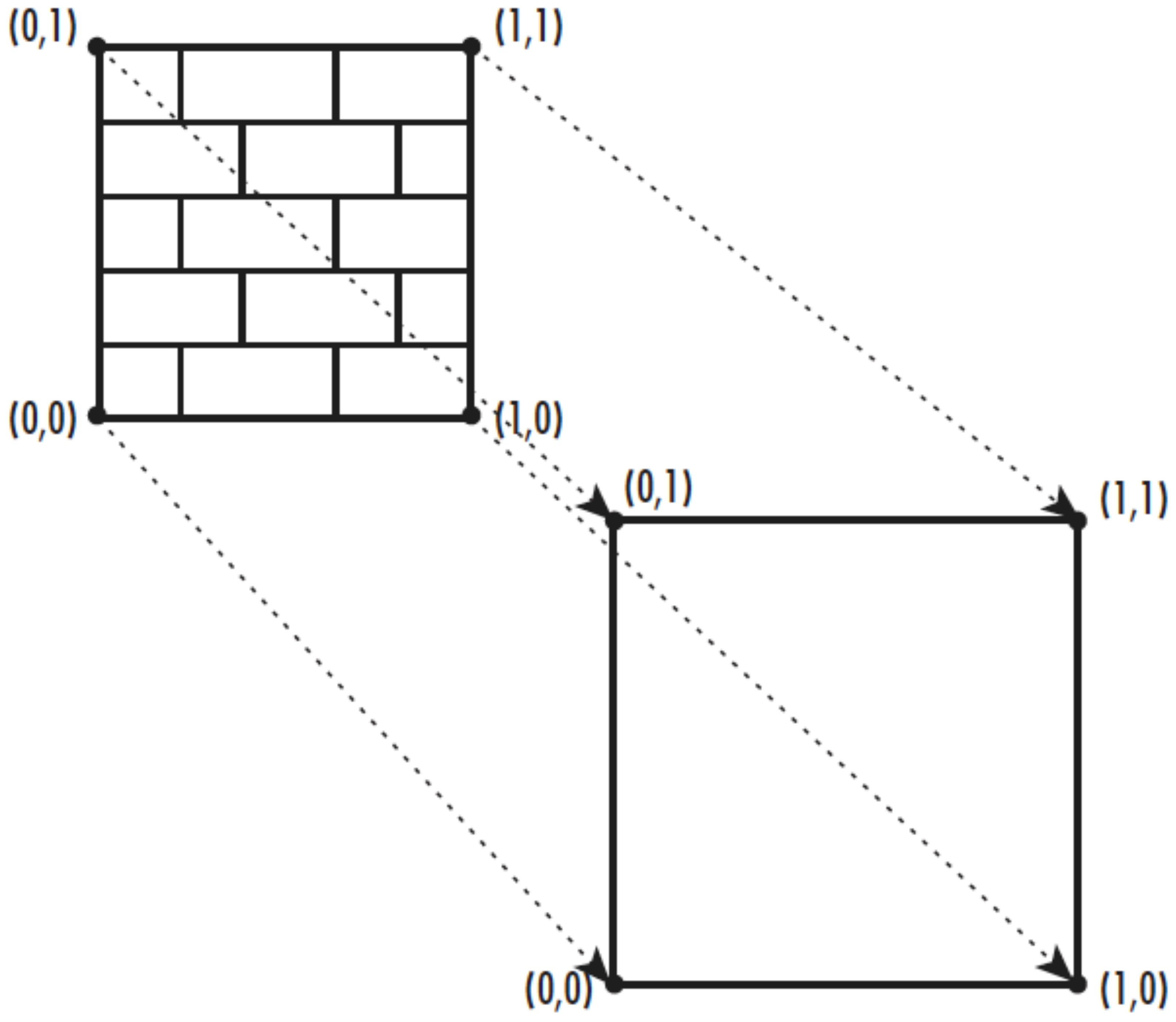


Image Formats

- We often work with images in a standard format (JPEG, TIFF, GIF)
  - How do we read/write such images with OpenGL?
- No support in OpenGL
  - OpenGL knows nothing of image formats
  - Can write readers/writers for some simple formats in OpenGL

Simple OpenGL Image Library (SOIL) - Features

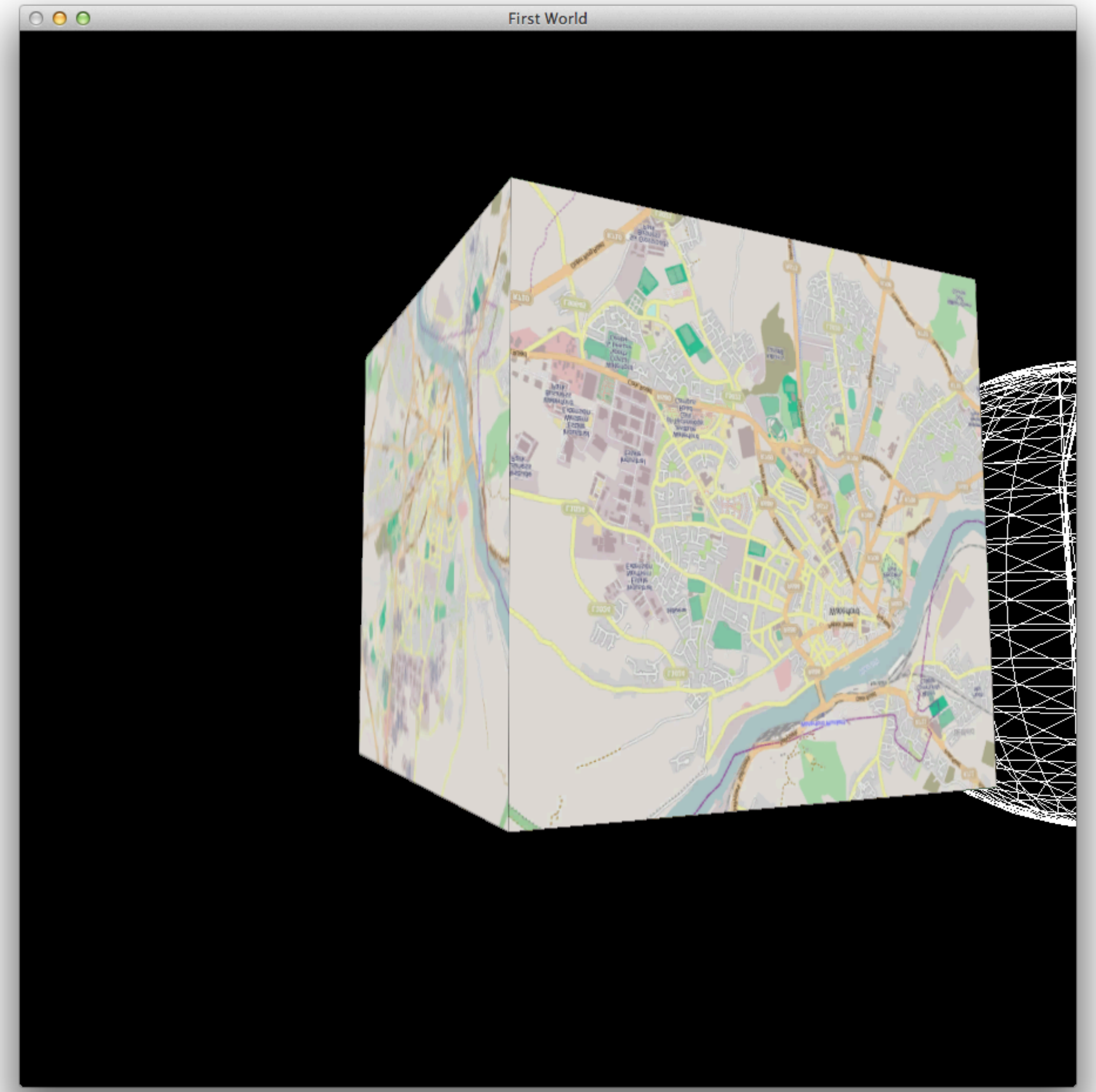
- Readable Image Formats:
  - BMP, PNG, JPG, TGA etc...
- Writeable Image Formats:
  - TGA, BMP etc...
- Can load an image file directly into a 2D OpenGL texture, optionally performing the following functions:
- Can generate a new texture handle, or reuse one specified
- Can automatically rescale the image to the next largest power-of-two size
- Can flip the image vertically
- No external dependencies
- Cross platform (Windows, *nix, Mac OS X)

http://www.lonesock.net/soil.html

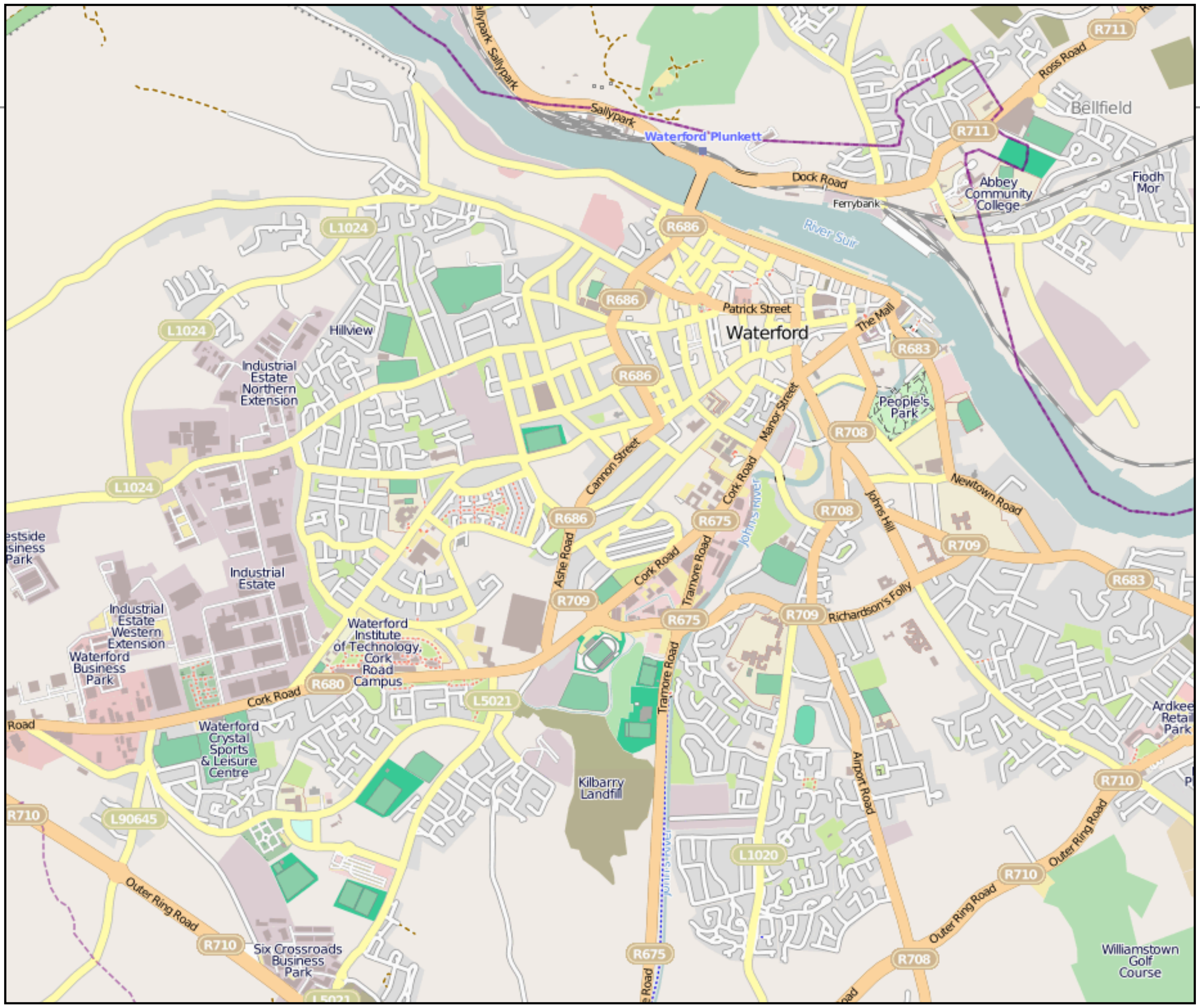
ImageCube

```
struct ImageCube: public  Actor
{
  int imageID;

  ImageCube();
  void render();
};
```



City.png



- Use SOIL to load as a ‘texture’

```
#include "SOIL.h"
...
imageID = loadTexture("city.png");
```

glBindTexture

```
struct ImageCube: public  Actor
{
  int imageID;

  ImageCube();
  void render();
};
```

- Enable Textures
- Set polygon mode to FILL
- ‘Bind’ a specific texture via its ID
- Generate the geometry

```
ImageCube::ImageCube()
{
  imageID = loadTexture("city.png");
}

void ImageCube::render()
{
  glPolygonMode(GL_FRONT,GL_FILL);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, imageID);

  glBegin( GL_QUADS);
  for (int i=0; i<6; i++)
  {
    drawFace(thevertices[i]);
  }
  glEnd();

  glDisable(GL_TEXTURE_2D);
  glPolygonMode(GL_FRONT,GL_LINE);
}
```

The geometry

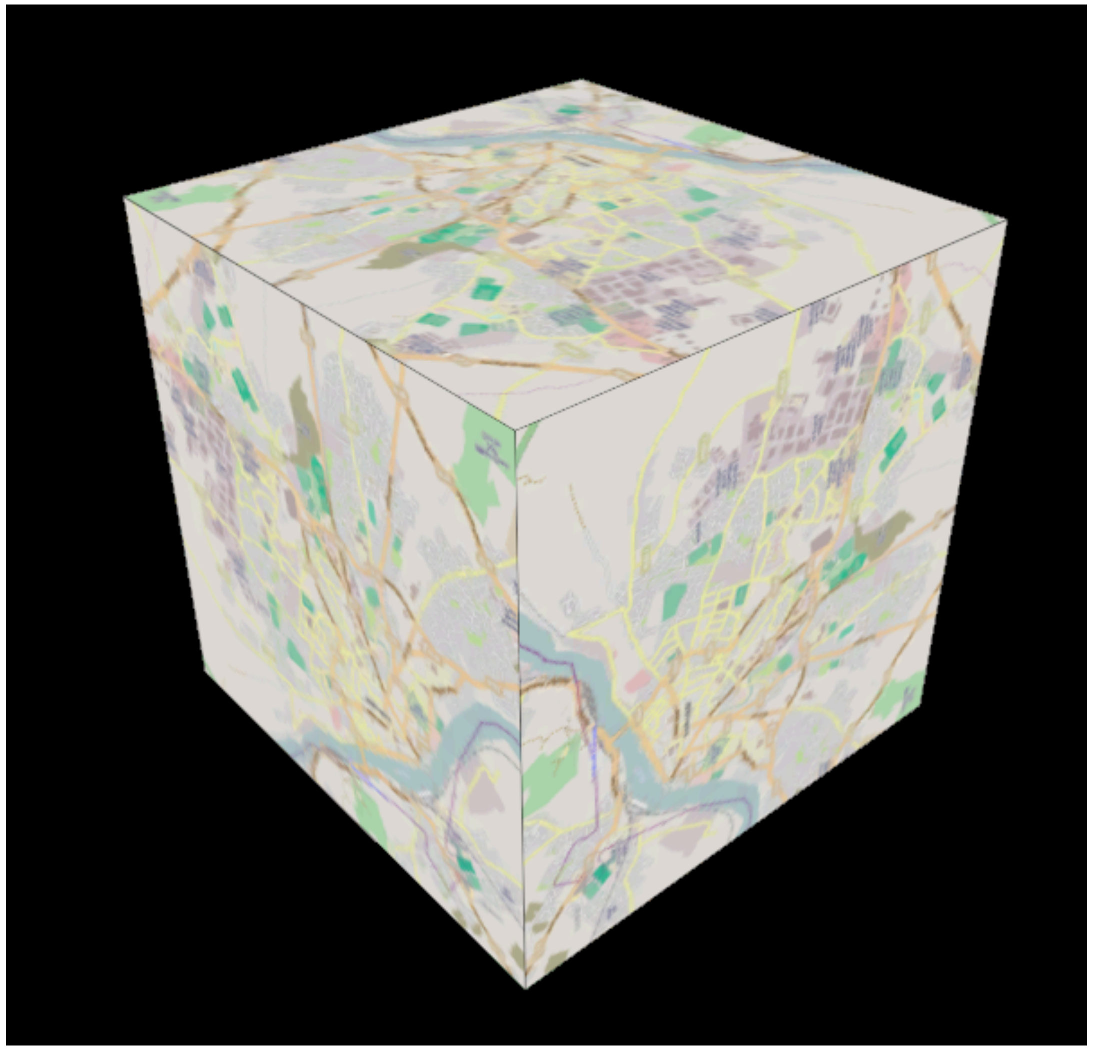
```
Vector3 thevertices[][6] =
{
  { Vector3(-1.0f, 1.0f, 1.0f), Vector3(-1.0f, -1.0f, 1.0f), Vector3( 1.0f, -1.0f, 1.0f), Vector3( 1.0f, 1.0f, 1.0f)  },
  { Vector3( 1.0f, 1.0f,-1.0f), Vector3( 1.0f, -1.0f,-1.0f), Vector3(-1.0f, -1.0f,-1.0f), Vector3(-1.0f, 1.0f, -1.0f) },
  { Vector3(-1.0f, 1.0f,-1.0f), Vector3(-1.0f,  1.0f, 1.0f), Vector3( 1.0f,  1.0f, 1.0f), Vector3( 1.0f, 1.0f, -1.0f) },
  { Vector3( 1.0f,-1.0f,-1.0f), Vector3( 1.0f, -1.0f, 1.0f), Vector3(-1.0f, -1.0f, 1.0f), Vector3(-1.0f,-1.0f, -1.0f) },
  { Vector3( 1.0f,-1.0f, 1.0f), Vector3( 1.0f, -1.0f,-1.0f), Vector3( 1.0f,  1.0f,-1.0f), Vector3( 1.0f, 1.0f,  1.0f) },
  { Vector3(-1.0f, 1.0f, 1.0f), Vector3(-1.0f, 1.0f, -1.0f), Vector3(-1.0f, -1.0f,-1.0f), Vector3(-1.0f, -1.0f, 1.0f) }
};
```

- Simple Unit Cube - Vertices only

```
void ImageCube::render()
{
  //...
  for (int i=0; i<6; i++)
  {
    drawFace(thevertices[i]);
  }
  //...
}
```

drawFace - glTextCoord

- Render and resize the image to fit the 6 faces of the cube.



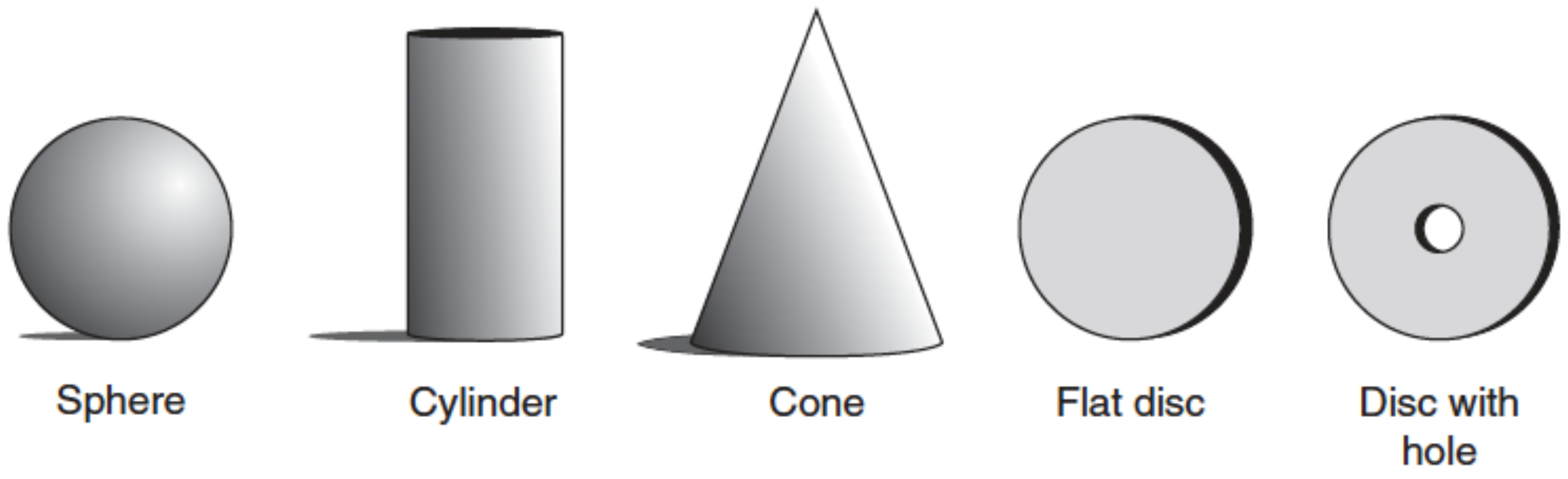
```
void drawFace(Vector3 vertices[])
{
  glTexCoord2f(0.0, 0.0);
  vertices[0].render();
  glTexCoord2f(0.0, 1.0);
  vertices[1].render();
  glTexCoord2f(1.0, 1.0);
  vertices[2].render();
  glTexCoord2f(1.0, 0.0);
  vertices[3].render();
}
```

in Scene:

```
  actorName = "imagecube";
  actors.insert(actorName, new ImageCube());
```

Quadrics

- The OpenGL Utility Library (GLU) that accompanies OpenGL contains a number of functions that render three quadratic surfaces.
- These quadric functions render spheres, cylinders,and disks.
- You can specify the radius of both ends of a cylinder. Setting one end’s radius to 0 produces a cone.



Setting Quadric States

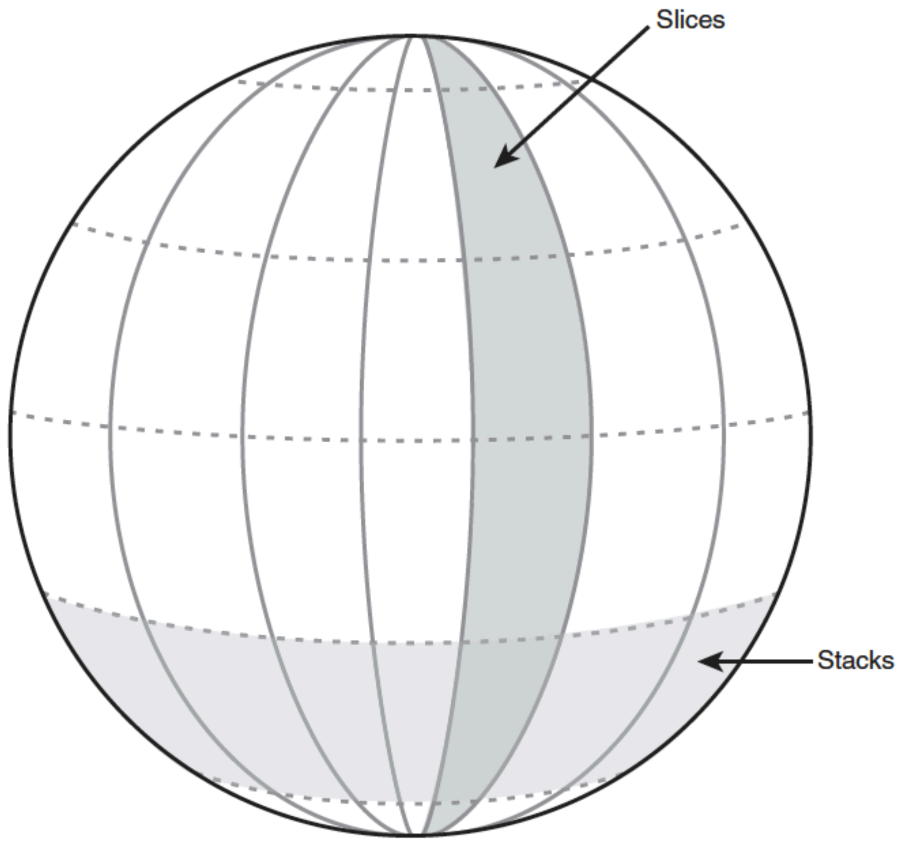
- The quadric surfaces can be drawn with some flexibility as to whether normals, texture coordinates, and so on are specified.
- Putting all these options into parameters to a sphere drawing function, for example, would create a function with an exceedingly long list of parameters that must be specified each time.
- Instead, the quadric functions use an object oriented model. Essentially, you create a quadric object and set its rendering state with one or more state setting functions

```
GLUquadric *qobj = gluNewQuadric();
// Set Quadric rendering Parameters
// Draw Quadric
gluDeleteQuadric(qobj);
```

Sphere

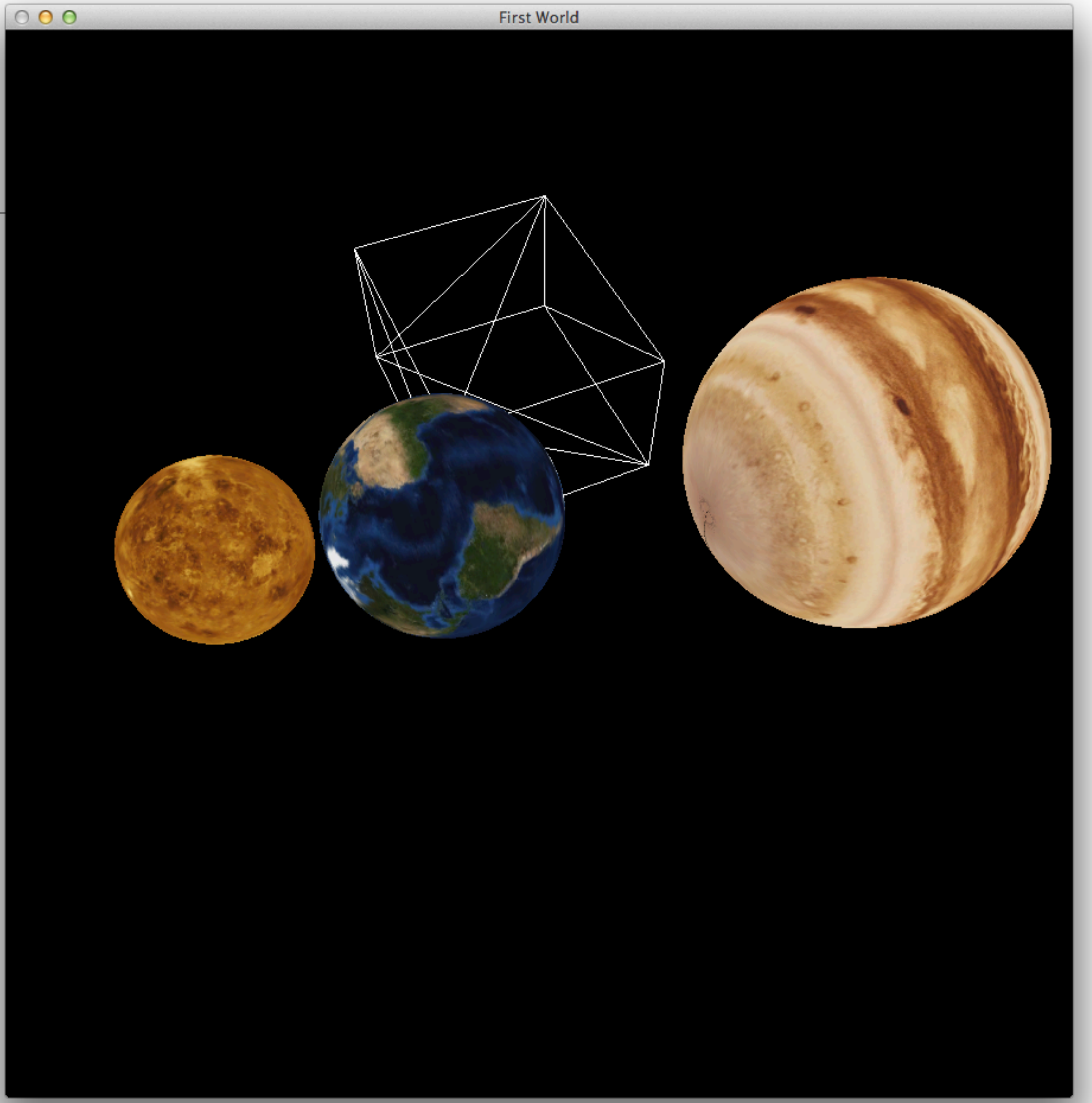
```
void gluSphere(GLUQuadricObj *obj, GLdouble radius, GLint slices, GLint stacks);
```

- The first parameter, obj, is just the pointer to the quadric object that was previously set up for the desired rendering state.
- The radius parameter is then the radius of the sphere, followed by the number of slices and stacks. Spheres are drawn with rings of triangle strips stacked from the bottom to the top.
- The number of slices specifies how many triangle sets (or quads) are used to go all the way around the sphere.
- You could also think of this as the number of lines of latitude and longitude around a globe.



Planets

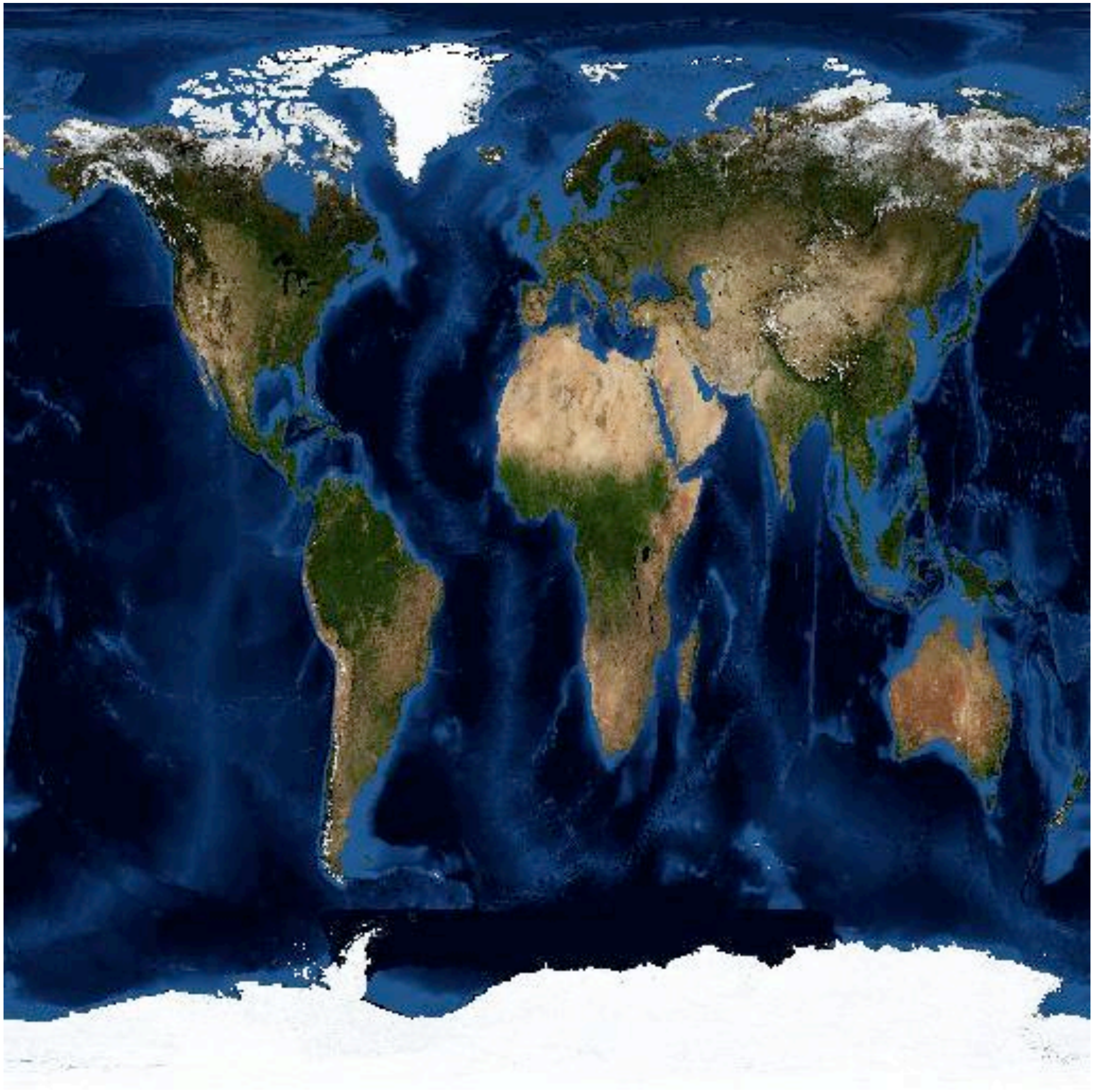
- Define three spheres
- Locate suitably ‘stretched’ image files
- Load and bind the texture
- Generate the appropriate texture co-ordinates
- Render



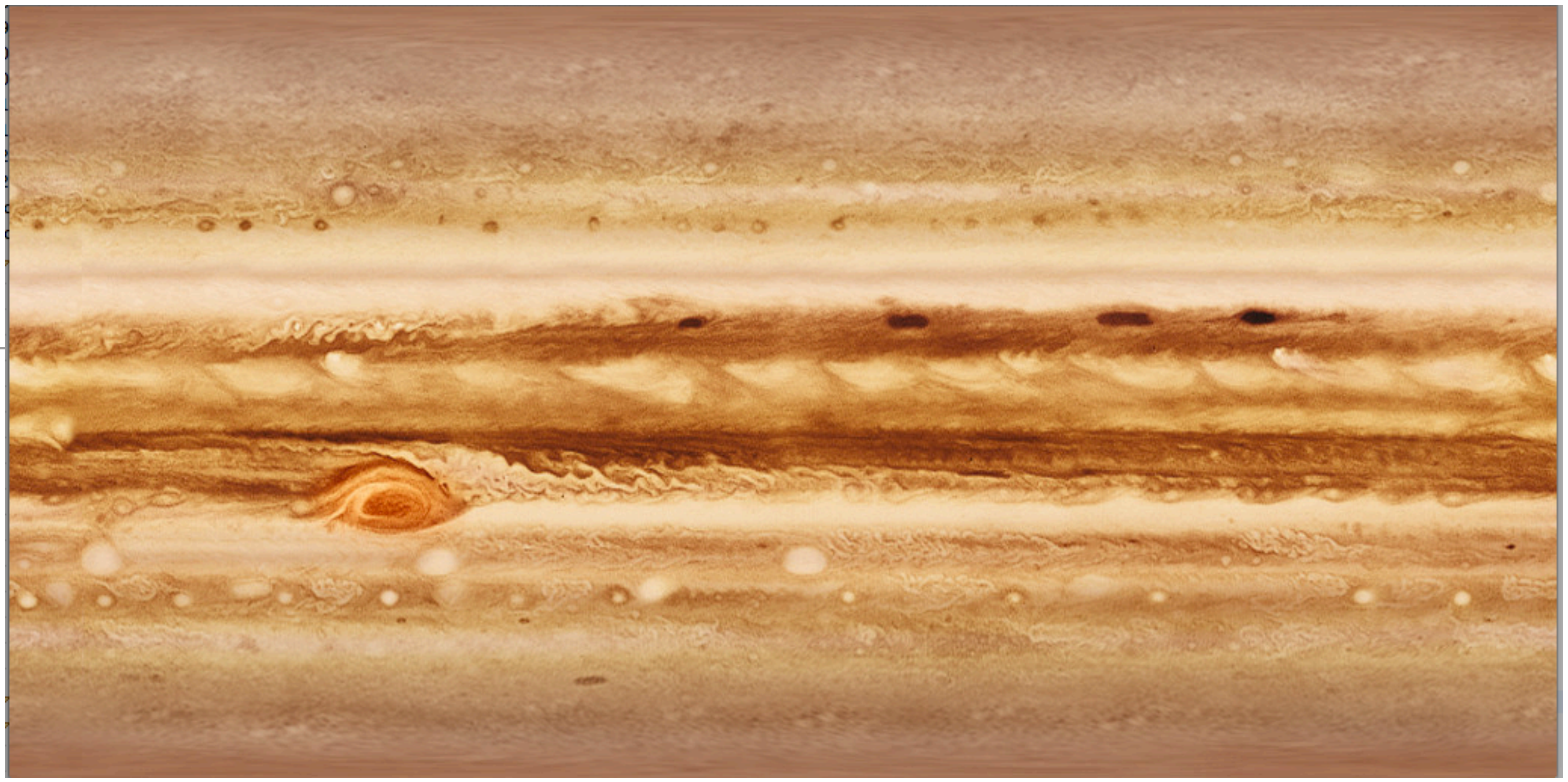
JHTs Planetary Emporium

http://planetpixelemporium.com/planets.html

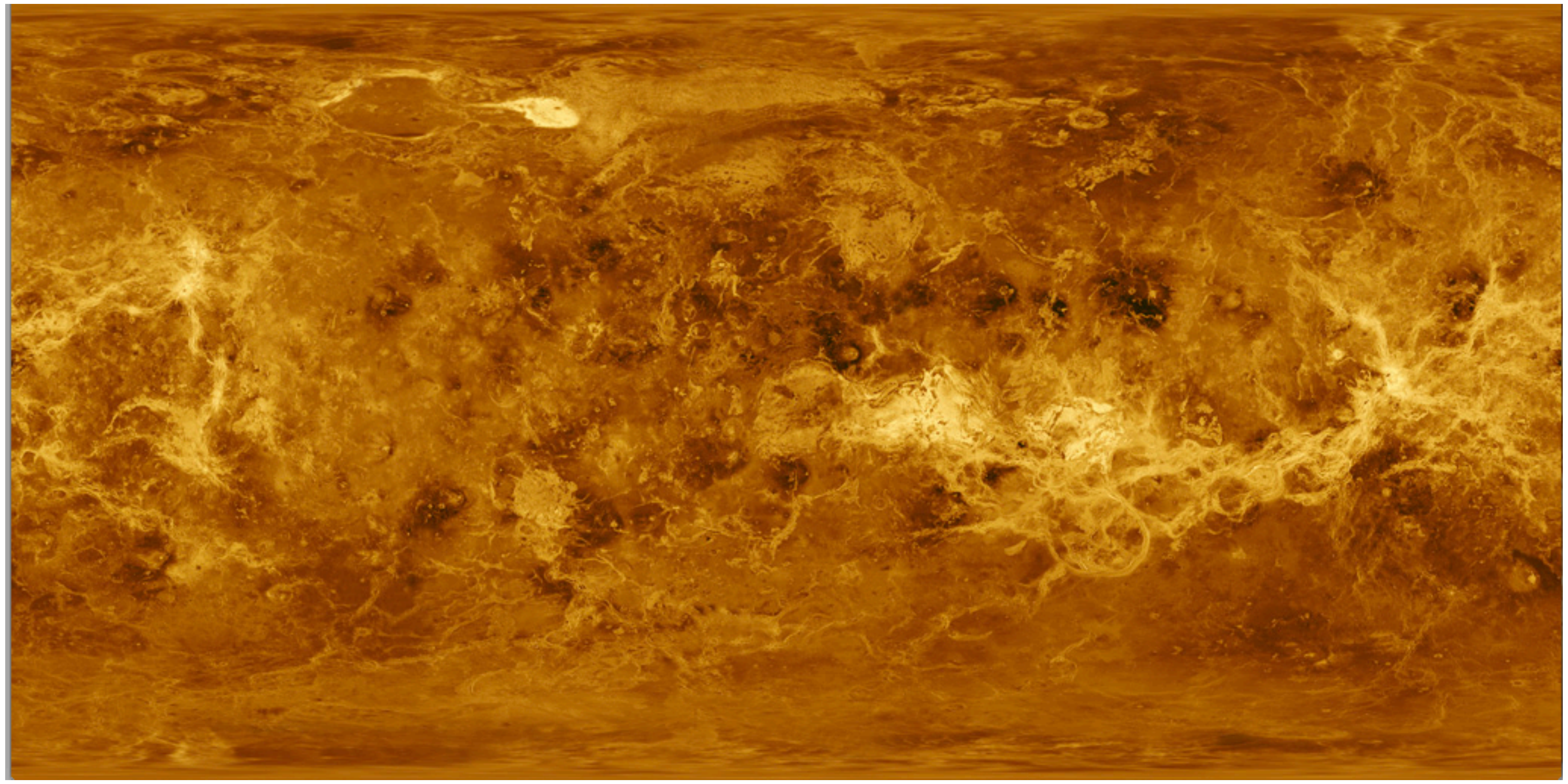
- Earth



- Jupiter



- Venus



Sphere

```
struct Sphere: public  Actor
{
  int     imageID;
  Vector3 position;

  Sphere(Vector3 position, std::string imagefile);
  void render();
};
```

```
Sphere::Sphere(Vector3 position, string imagefilename)
: position (position)
{
  imageID = loadTexture(imagefilename);
}

void Sphere::render()
{
  glPolygonMode(GL_FRONT,GL_FILL);
  glEnable(GL_TEXTURE_2D);

  GLUquadric *qobj = gluNewQuadric();
  gluQuadricTexture(qobj,GL_TRUE);
  glBindTexture(GL_TEXTURE_2D, imageID);

  gluSphere(qobj,1,50,50);

  gluDeleteQuadric(qobj);

  glPolygonMode(GL_FRONT,GL_LINE);
  glDisable(GL_TEXTURE_2D);
}
```