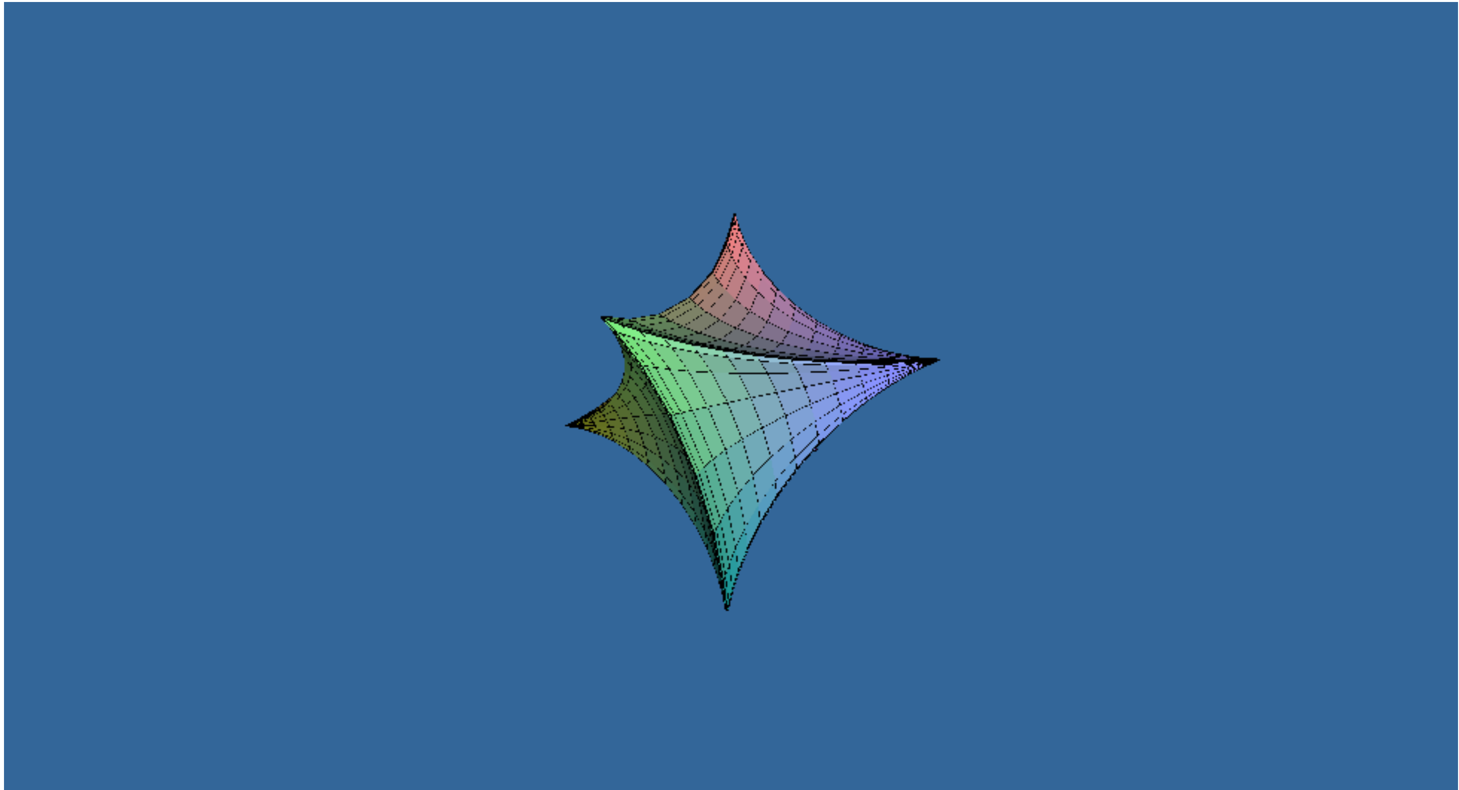
Advanced Graphics



OpenGL

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Today’s technologies

Java
- Common, re-usable language; extremely well-designed
- Steadily increasing popularity in industry
- Weak but evolving 3D support

C++
- Long-established language
- Long history with OpenGL
  - Technically C has the long history. C++ never really improved it.
- Long history with DirectX
- Losing popularity in some fields (finance, web) but still strong in others (games, medical)

OpenGL
- Open source with many implementations
- Extraordinarily well-designed, old, but still evolving
- Fairly cross-platform

DirectX/Direct3d
- Less well-designed
- Microsoft™ only
  - DX 10 *requires* Vista!
- But! Dependable updates…

Java3D
- Poor cross-platform support (surprisingly!)
- Available by GPL; community-developed

OpenGL

OpenGL is…

- hardware-independent
- operating system independent
- vendor neutral

OpenGL is a *state-based* renderer

- set up the state, then pass in data: data is modified by existing state
- very different from the OOP model, where data would carry its own state

OpenGL

OpenGL is platform-independent, but implementations are platform-specific and often rely on native libraries

- Great support for Windows, Mac, linux, etc
- Support for mobile devices with OpenGL-ES
    - Android, iPhone, Symbian OS

Accelerates common 3D graphics operations

- Clipping (for primitives)
- Hidden-surface removal (Z-buffering)
- Texturing, alpha blending (transparency)
- NURBS and other advanced primitives (GLUT)

OpenGL in Java: *JOGL*

JOGL is the Java binding for OpenGL.

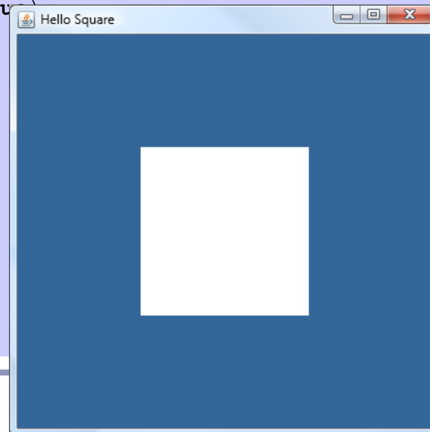
- JOGL apps can be deployed as applications or as applets.
  - This means that you can embed 3D in a web page.
    - (If the user has installed the latest Java, of course.)
    - Admittedly, applets are somewhat "1998".

Using JOGL:

- Wiki: http://en.wikipedia.org/wiki/Java_OpenGL
- You can download JOGL from http://opengl.j3d.org/You can download JOGL from http://opengl.j3d.org/ and http://kenai.com/projects/jogl/
- To deploy an embedded applet, you'll use Sun's JNLP wrappers, which provide signed applets wrapped around native JOGL binaries.

A quick intro to JOGL: *Hello Square*

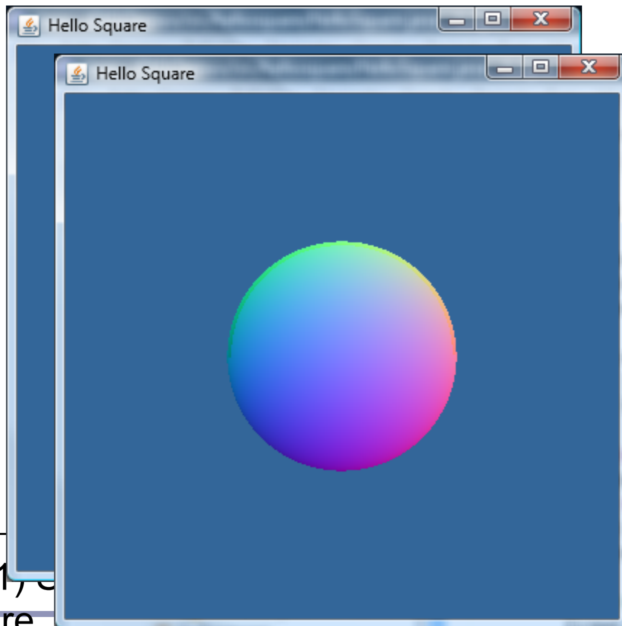
```
public class HelloSquare {
  public static void main(String[] args) {
    new Thread() {
      public void run() {
        Frame frame = new Frame("Hello Square");
        GLCanvas canvas = new GLCanvas();

        // Setup GL canvas
        frame.add(canvas);
        canvas.addGLEventListener(new Renderer());

        // Setup AWT frame
        frame.setSize(400, 400);
        frame.addWindowListener(new WindowAdapter(){
          public void windowClosing(WindowEvent e) {
            System.exit(0);
          }
        });
        frame.setVisible(true);

        // Render loop
        while(true) {
          canvas.display();
        }
      }
    }.start();
  }
}
```




```
public class Renderer implements GLEventListener {
  public void init(GLAutoDrawable glDrawable) {
    final GL gl = glDrawable.getGL();
    gl.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);
  }

  public void display(GLAutoDrawable glDrawable) {
    final GL gl = glDrawable.getGL();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glLoadIdentity();
    gl.glTranslatef(0, 0, -5);

    gl.glBegin(GL.GL_QUADS);
      gl.glVertex3f(-1, -1,  0);
      gl.glVertex3f( 1, -1,  0);
      gl.glVertex3f( 1,  1,  0);
      gl.glVertex3f(-1,  1,  0);
    gl.glEnd();
  }

  public void reshape(GLAutoDrawable glDrawable,
 int x, int y, int width, int height) {
    final GL gl = glDrawable.getGL();
    final float h = (float)width / (float)height;

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    (new GLU()).gluPerspective(50, h, 1, 1000);
    gl.glMatrixMode(GL.GL_MODELVIEW);
  }
}
```

A simple parametric surface in JOGL

```
public void vertex(GL gl,
  float x, float y, float z) {
  gl.glColor3f(
    (x+1)/2.0f,
    (y+1)/2.0f,
    (z+1)/2.0f);
  gl.glVertex3f(x, y, z);
}
```



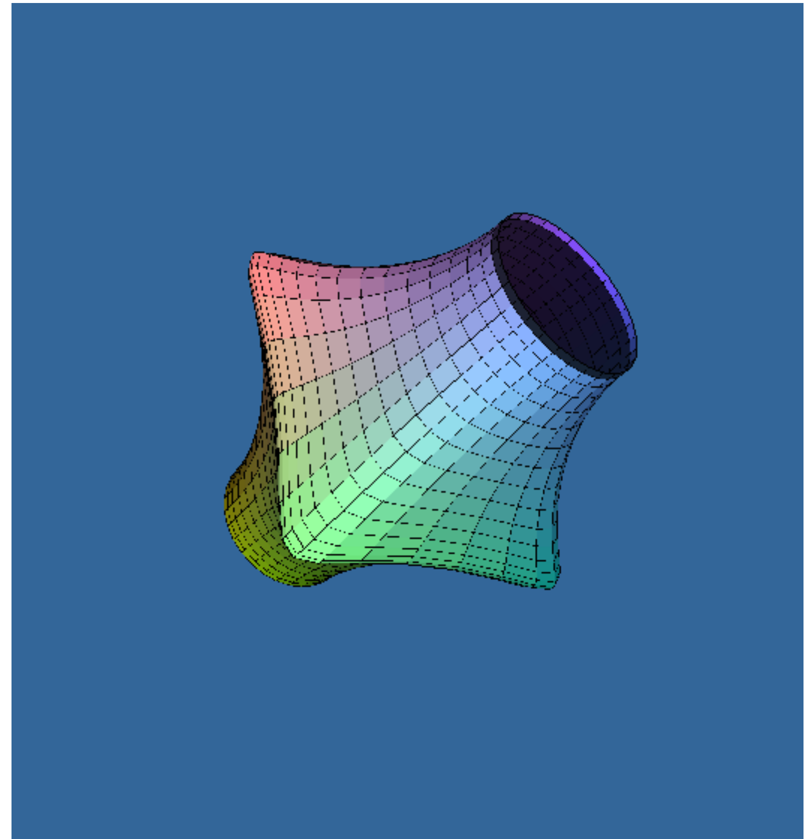
1) C
square

2) Parametric
sphere

Animating a parametric surface

The animation at right shows the linear interpolation between four parametric surface functions.

- Colors are by XYZ.
- The code is online, and pretty simple—please play with it



Behind the scenes

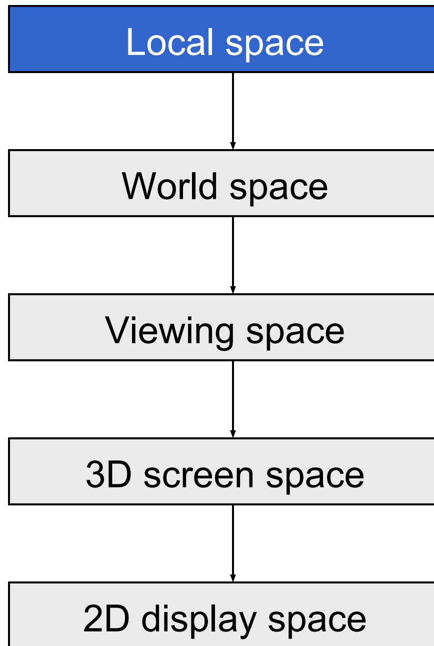
Two players:

- The CPU, your processor and friend
- The *GPU* (*Graphical Processing Unit*) or equivalent software

The CPU passes streams of vertices and of data to the GPU.

- The GPU processes the vertices according to the *state* that has been set; here, that state is "every four vertices is one quadrilateral polygon".
- The GPU takes in streams of vertices, colors, texture coordinates and other data; constructs polygons and other primitives; then draws the primitives to the screen pixel-by-pixel.
- This process is called the *rendering pipeline*.

Anatomy of a rendering pipeline

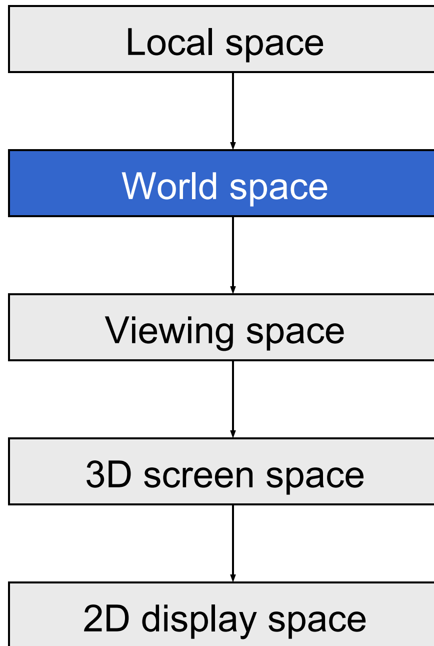


1) Geometry is defined in *local space*. The vertices and coordinates of a surface are specified relative to a local basis and origin.

This encourages re-use and replication of geometry; it also saves the tedious math of storing rotations and other transformations within the vertices of the shape itself.

This means that changing the position of a highly complex object requires only changing a 4x4 matrix instead of recalculating all vertex values.

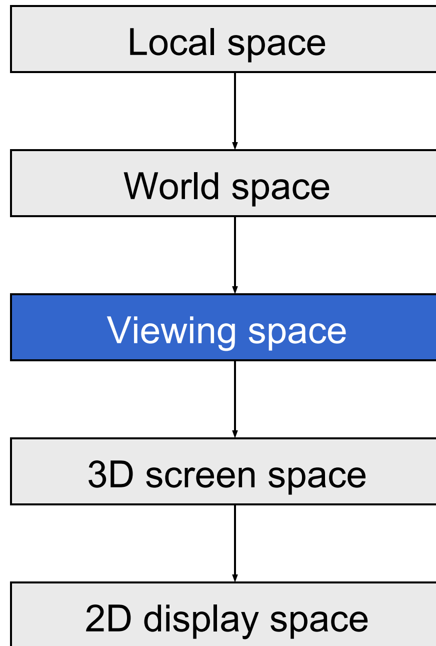
Anatomy of a rendering pipeline



2) The pipeline transforms vertices and surface normals from *local* to *world* space.

A series of matrices are concatenated together to form the single transformation which is applied to each vertex. The rendering engine (e.g., OpenGL) is responsible for associating the state that transforms each group of vertices with the actual vertex values themselves.

Anatomy of a rendering pipeline



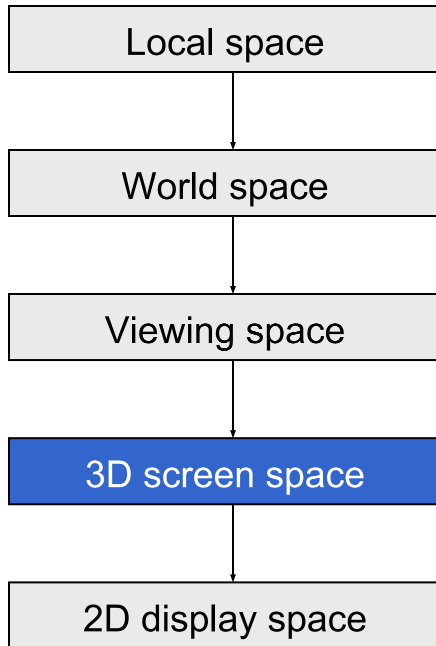
3) Rotate and translate the geometry from *world* space to *viewing* or *camera* space.

At this stage, all vertices are positioned relative to the point of view of the camera. (The world really does revolve around you!)

For example, a cube at (10,000, 0, 0) viewed from a camera (9,999, 0, 0) would now have relative position (1, 0, 0). Rotations would have similar effect.

This makes operations such as clipping and hidden-object removal much faster.

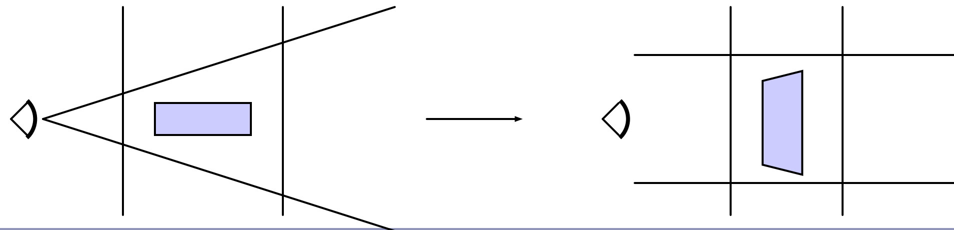
Anatomy of a rendering pipeline



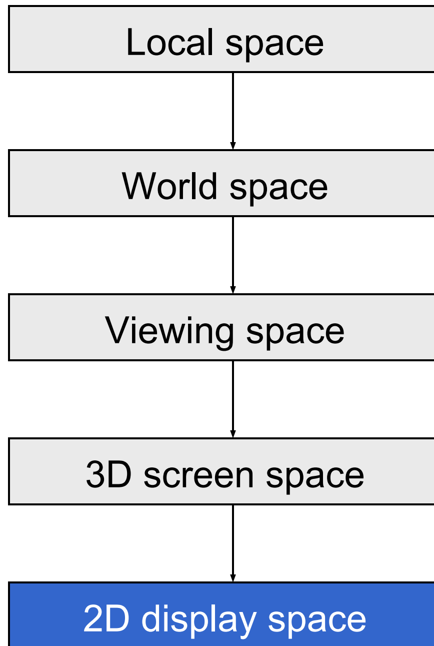
4) Perspective: Transform the viewing frustrum into an axis-aligned box with the near clip plane at z=0 and the far clip plane at z=1. Coordinates are now in *3D screen space*.

This transformation is *not affine*: angles will distort and scales change.

Hidden-surface removal can be accelerated here by clipping objects and primitives against the viewing frustrum. Depending on implementation this clipping could be before transformation or after or both.



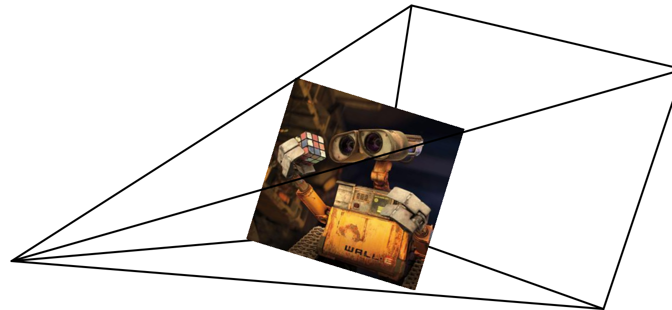
Anatomy of a rendering pipeline



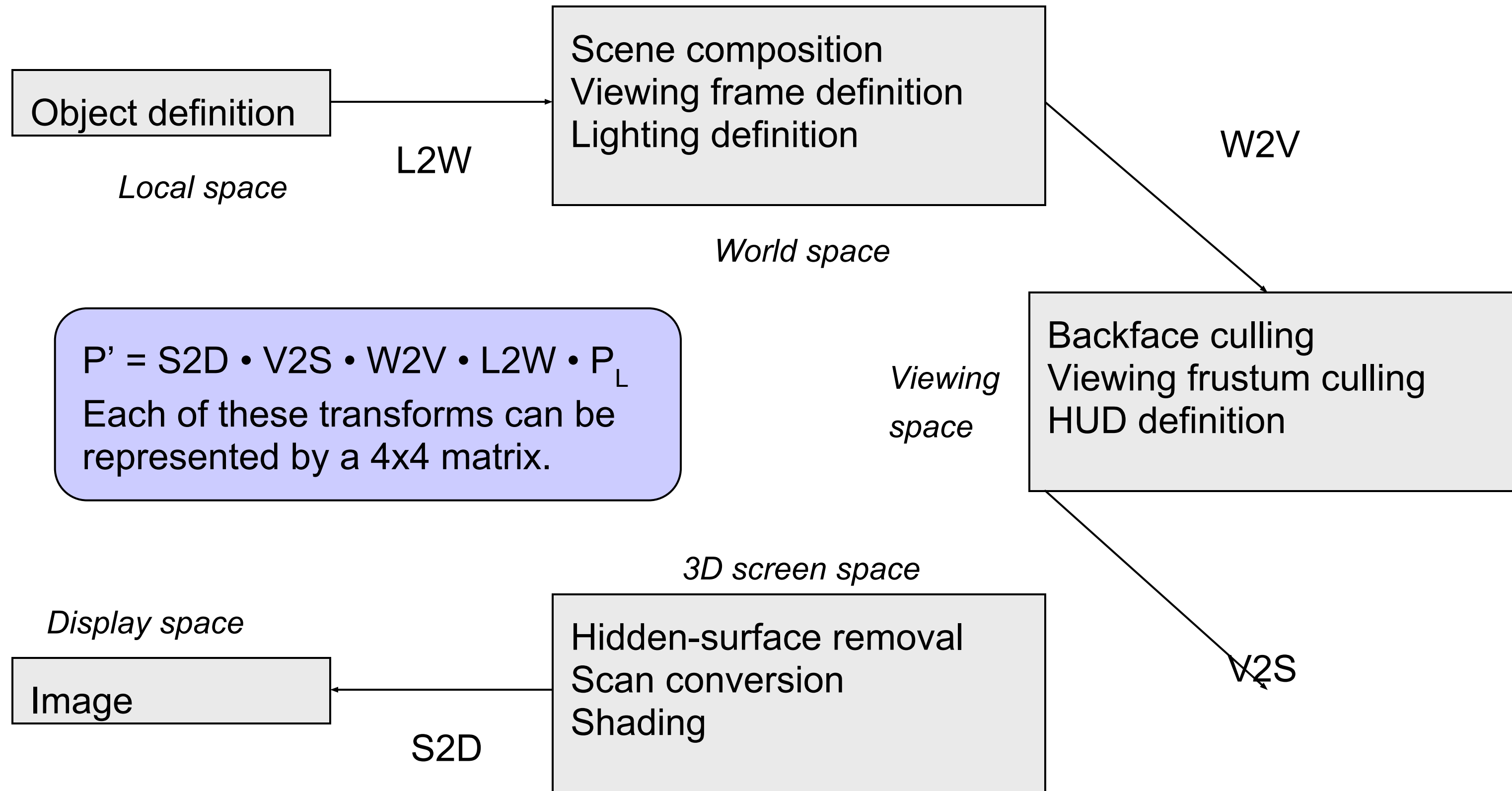
5) Collapse the box to a plane. Rasterize primitives using Z-axis information for depth-sorting and hidden-surface-removal.

Clip primitives to the screen.

Scale raster image to the final raster buffer and rasterize primitives.



Recap: sketch of a rendering pipeline



OpenGL's matrix stacks

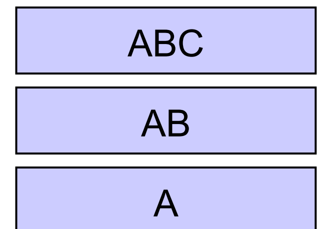
OpenGL uses *matrix stacks* to store stacks of matrices, where the topmost matrix is (usually) the product of all matrices below.

- This allows you to build a local frame of reference—local space—and apply transforms within that space.

Remember: matrix multiplication is associative but not commutative.

- $ABC = A(BC) = (AB)C \neq ACB \neq BCA$

Pre-multiplying matrices that will be used more than once is faster than multiplying many matrices every time you render a primitive.



OpenGL's matrix stacks

GL has three matrix stacks:

- **Modelview** – positioning things relative to other things
- **Projection** – camera transforms
- **Texture** – texture-mapping transformations

You choose your current matrix with *`glMatrixMode()`*; this sets the state for all following matrix operations.

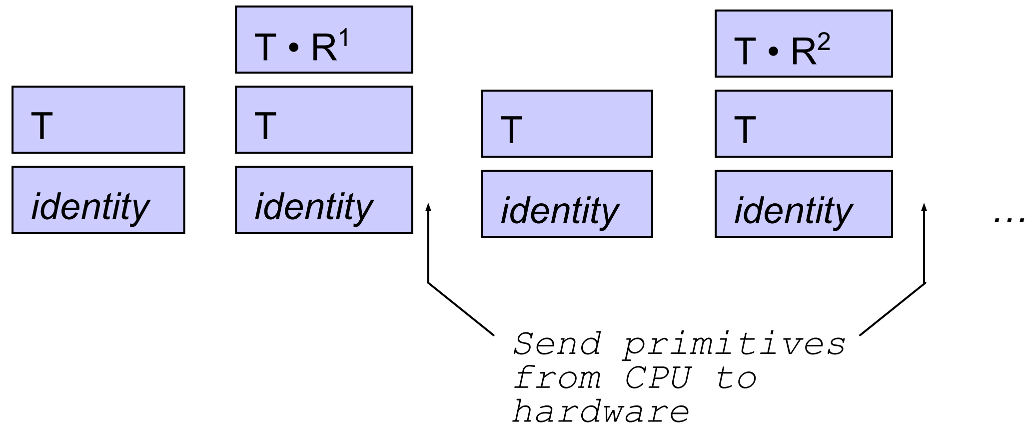
Each time you call `glTranslate()`, `glRotate()`, etc., these commands modify the current topmost matrix on the current stack.

If you want to make local changes that only have limited effect, you use `glPushMatrix()` to push a new copy of your current matrix onto the top of the stack; then you modify it freely and, when done, call `glPopMatrix()`.

Matrix stacks and scene graphs

Matrix stacks are designed for nested relative transforms.

```
glPushMatrix();
  glTranslatef(0,0,-5);
  glPushMatrix();
    glRotatef(45,0,1,0);
    renderSquare();
  glPopMatrix();
  glPushMatrix();
    glRotatef(-45,0,1,0);
    renderSquare();
  glPopMatrix();
glPopMatrix();
```



Rendering simple primitives

GL's state machine applies its state to each vertex in sequence.

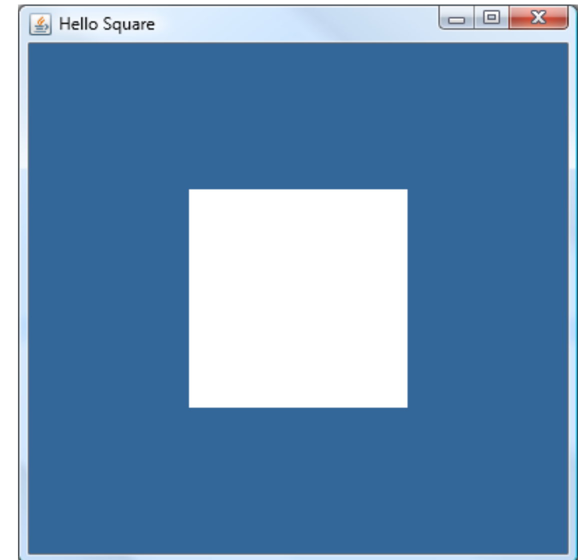
To render simple primitives, tell GL what kind of primitive to render:

- `glBegin(GL_LINES)`
- `glBegin(GL_LINE_STRIP)`
- `glBegin(GL_TRIANGLES)`
- `glBegin(GL_QUADS)`
- `glBegin(GL_TRIANGLE_STRIP)`
- And several others

After calling `glBegin()`, you can call `glVertex()` repeatedly, passing in triples (or quads) of floats (or doubles) which are interpreted as positions in the context of the current rendering state.

- GL is very flexible about data sizes and data types

When you're done, call `glEnd()`. Your primitives will now be rasterized.




```
glBegin(GL.GL_QUADS);
glVertex3f(-1, -1,  0);
glVertex3f( 1, -1,  0);
glVertex3f( 1,  1,  0);
glVertex3f(-1,  1,  0);
glEnd();
```

Rendering primitives in a slightly less painfully inefficient manner

Instead of sending each vertex individually, send them *en masse*:

```
GLfloat vertices[] = {...};                    // Set up triples of floats
glEnableClientState(GL_VERTEX_ARRAY);          // We'll be rendering a vertex array
glVertexPointer(3, GL_FLOAT, 0, vertices);     // Which vertices we'll be rendering
glDrawArrays(GL_QUADS, 0, numVerts);           // Render
glDisableClientState(GL_VERTEX_ARRAY);         // Stop rendering vertex arrays
```

Using `glDrawArrays()` we can avoid the overhead of a huge number of `glVertex()` calls.

Rendering primitives in a way that's really quite efficient, actually

`glDrawArrays()` takes a bulk list of vertices, but it still sends every vertex to the GPU once for every triangle or quad that uses it.

If your surface repeats the same vertex more than once, you can use `glDrawElements()` instead. `glDrawElements()` acts like `glDrawArrays()` but takes an additional list of indices into the array.

- Now you'll pass down each vertex exactly once, referencing its integer index multiple times.

```
GLfloat vertices[] = {...};                         // Set up triples of floats
GLubyte indices[] = {...};                          // Set up vertex indices list
glEnableClientState(GL_VERTEX_ARRAY);               // We'll be rendering a vertex
array
glVertexPointer(3, GL_FLOAT, 0, vertices);          // Which vertices we'll be
rendering
glDrawElements(GL_QUADS, numVerts, GL_UNSIGNED_BYTE, indices);     // Render
glDisableClientState(GL_VERTEX_ARRAY);              // Stop rendering vertex arrays
```

Camera control in OpenGL

OpenGL has two stacks that apply to geometry being rendered: **Modelview** and **Projection**.

- The values atop these two stacks are concatenated to transform each vertex from local to world to screen space.
- You set up perspective on the Projection stack
- You position your scene in world co-ordinates on the Modelview stack

You can position your camera on either stack; it's just another transform

- GL's utility library, glu, provides several convenient utility methods to set up a perspective view:
  - `gluLookAt`
  - `gluPerspective`
  - `gluOrtho`, etc

By default your camera sits at the origin, pointing down the negative Z axis, with an up vector of `(0,1,0)`.

I usually set my camera position on the Modelview matrix stack.

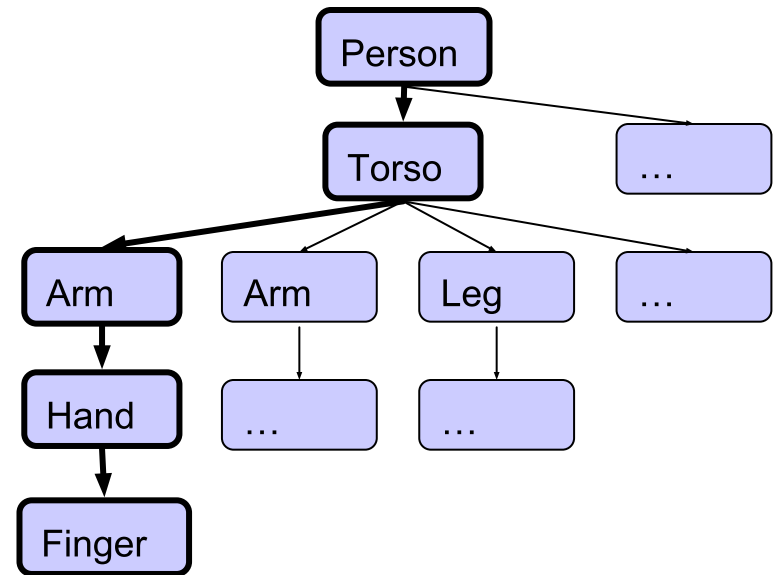
```
gl.glMatrixMode(GL.GL_MODELVIEW);              // Switch to model
stack
gl.glLoadIdentity();                            // Reset to identity
gl.glTranslated(0.0, 0.0, -cameraDistance);     // Slide the model away
gl.glMultMatrixd(cameraTransformMatrix, 0);     // Spin the model
```

Scene graphs

A *scene graph* is a tree of scene elements where a child's transform is relative to its parent.

The final transform of the child is the ordered product of all of its ancestors in the tree.

OpenGL's matrix stack and depth-first traversal of your scene graph: two great tastes that go great together!



$$M_{fingerToWorld} = (M_{person} \cdot M_{torso} \cdot M_{arm} \cdot M_{hand} \cdot M_{finger})$$

Your scene graph and you

A common optimization derived from the scene graph is the propagation of *bounding volumes*.

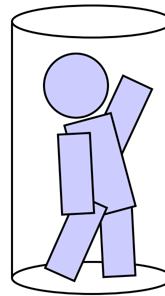
- These take many forms: bounding spheres, axis-aligned bounding boxes, oriented bounding boxes…

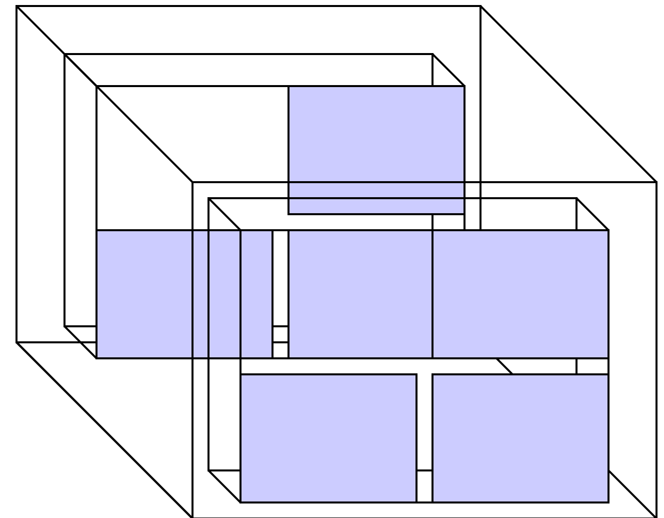
Nested bounding volumes allow the rapid culling of large portions of geometry

- Test against the bounding volume of the top of the scene graph and then work down.

Great for…

- Collision detection between scene elements
- Culling before rendering
- Accelerating ray-tracing





Your scene graph and you

Many 2D GUIs today favor an event model in which events 'bubble up' from child windows to parents. This is sometimes mirrored in a scene graph.

- Ex: a child changes size, which changes the size of the parent's bounding box
- Ex: the user drags a movable control in the scene, triggering an update event

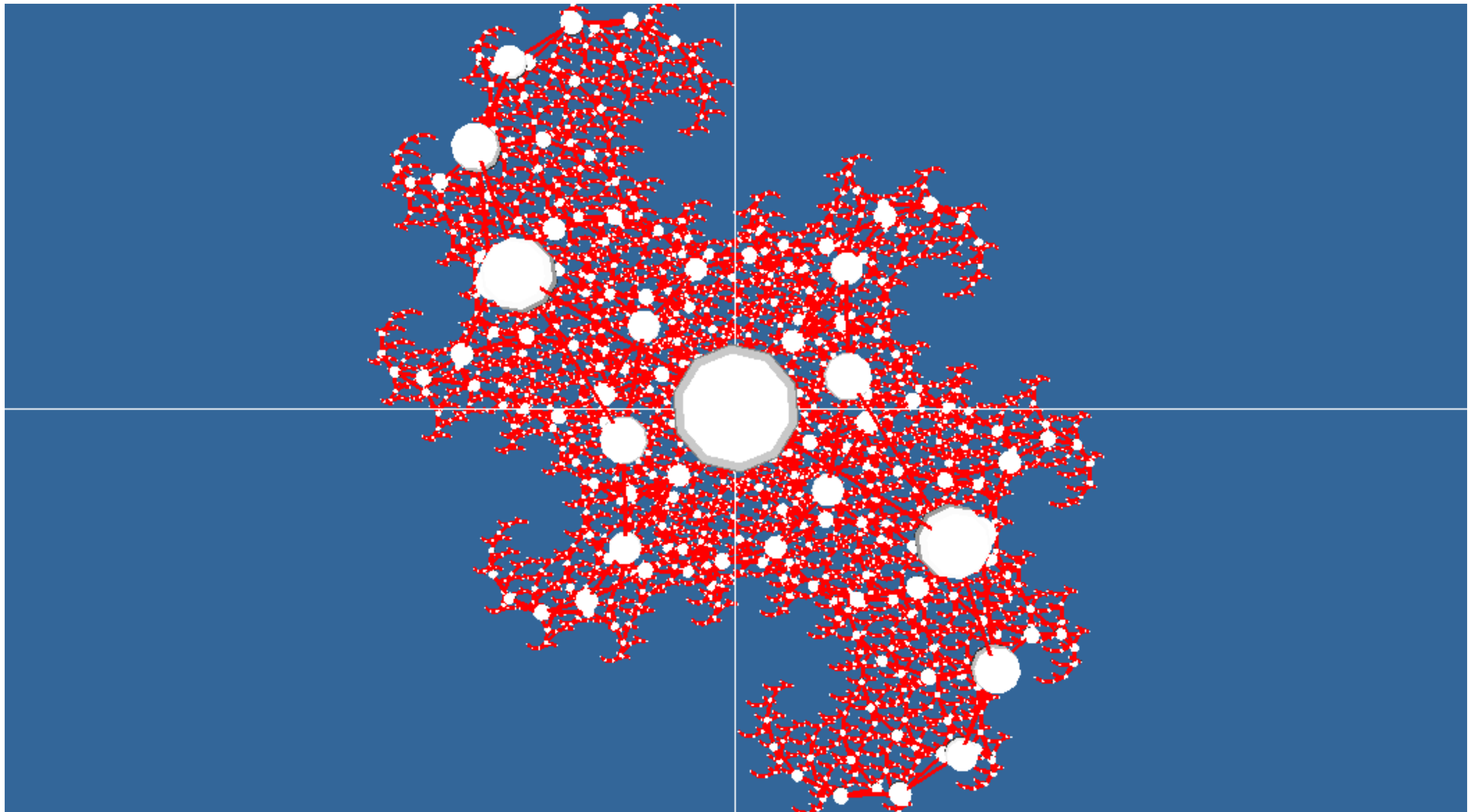
If you do choose this approach, consider using the *model/ view/ controller* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.

- For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
- Always separate logic from representation.

Hierarchical modeling in action

```
void renderLevel(GL gl, int level, float t) {
  gl.glPushMatrix();
  gl.glRotatef(t, 0, 1, 0);
  renderSphere(gl);
  if (level > 0) {
    gl.glScalef(0.75f, 0.75f, 0.75f);
    gl.glPushMatrix();
      gl.glTranslatef(1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    gl.glPopMatrix();
    gl.glPushMatrix();
      gl.glTranslatef(-1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    gl.glPopMatrix();
  }
  gl.glPopMatrix();
}
```

Hierarchical modeling in action



Mobile OpenGL: *OpenGL-ES*

GL has been ported, slightly redux, to mobile platforms:

- Symbian
- Android
- iPhone
- Windows Mobile

Currently two flavors:

- 1.x for ‘fixed function’ hardware
- 2.x for ‘programmable’ hardware (with shader support)
  - Chips with built-in shader support are now available; effectively GPUs for cell phones

Mobile OpenGL: *OpenGL-ES*

Key traits of OpenGL-ES:

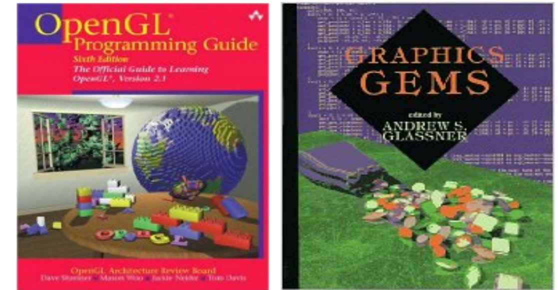
- Very small memory footprint
- Very low power consumption
- Smooth transitions from software rendering on low-end devices to hardware rendering on high-end; the developer should never have to worry
- Surprisingly wide-spread industry adoption

OpenGL-ES 2.0+ emphasize *shaders* over software running on the phone's processor. Shaders move processing from the device CPU to the peripheral GPU--mobile parallel processing.

Augmented reality with portable 3D

"ARhrrrr", an 'augmented reality' game concept from the Georgia Tech Augmented Reality Lab

Source: http://www.youtube.com/watch?v=cNu4CluFOcw



Recommended reading

The OpenGL Programming Guide

- Some folks also favor *The OpenGL Superbible* for code samples and demos
- There's also an OpenGL-ES reference, same series

The *Graphics Gems* series by Glassner et al

- All the maths you've already forgotten

The NeonHelium online OpenGL tutorials

- http://nehe.gamedev.net/