

Introduction to OpenGL

Lecturer:

Carol O'Sullivan

Professor of Visual Computing

Carol.OSullivan@cs.tcd.ie

Course www:

<http://isg.cs.tcd.ie/cosulliv/>

Some slides are borrowed from the University of Melbourne,
<http://www.cs.mu.oz.au/380/>

Overview

- OpenGL background
- OpenGL
- OpenGL pipeline
- OpenGL conventions, drawing
- Event loop, callback registration
- OpenGL primitives, OpenGL objects
- Books, resources, recommended reading

OpenGL Background

- OpenGL = Open Graphics Library
- Developed at Silicon Graphics (SGI)
- Successor to IrisGL
- Cross Platform
 - (Win32, Mac OS X, Unix, Linux)
- Only does 3D Graphics. No Platform Specifics
 - (Windowing, Fonts, Input, GUI)
- OpenGL 3.2 released on August 3, 2009
- Two Libraries
 - GL (Graphics Library)
 - GLU (Graphics Library Utilities)
 - Uses OpenGL to provide higher level drawing routines

Other Graphics Technology

- Low Level Graphics
- OpenGL
- Scene Graphs, BSPs
 - OpenSceneGraph, Java3D, VRML, PLIB
- DirectX (Direct3D)
- Can mix some DirectX with OpenGL (e.g OpenGL and DirectInput in Quake III)

Platform Specifics

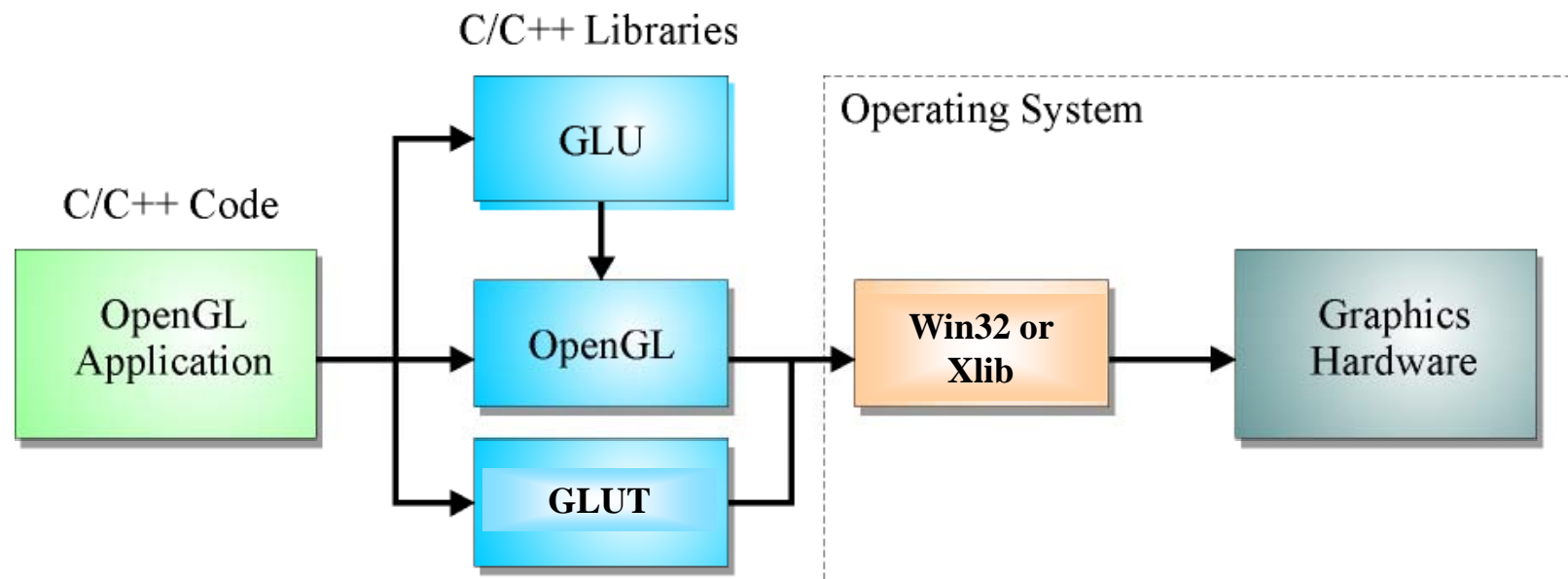
- Platform Specific OpenGL Interfaces
 - Windows (WGL)
 - X11 (GLX)
 - Motif (GLwMwidget)
 - Mac OS X (CGL/AGL/NSOpenGL)
- Cross Platform OpenGL Interface
 - Qt (QGLWidget, QGLContext)
 - Java (JOGL)
 - GLUT (GL Utility Library)

OpenGL

- OpenGL is a *software interface* to a graphics system implemented in hardware or software.
- About 250 distinct commands.
- It is (theoretically at least) *device independent*.
- OpenGL uses a *client-server* model, where client and server need not reside on the same machine.
- Default language is C/C++.
- To the programmer OpenGL behaves like a state machine.
- The actual drawing operations are performed by the underlying windows system or *accelerated graphics hardware* (where available, e.g. Nvidia, ATI, SGI etc).

OpenGL

- Programmer's model:



OpenGL

- OpenGL is *interactive* and *dynamic* and therefore we must handle interaction from the user \Rightarrow *event driven model*
- The **GL** library is the core OpenGL system:
 - modeling, viewing, lighting, clipping
- The **GLU** library (GL Utility) simplifies common tasks:
 - creation of common objects (e.g. spheres, quadrics)
 - specification of standard views (e.g. perspective, orthographic)
- The **GLUT** library (GL Utility Toolkit) provides the interface with the windowing system.
 - window management, menus, mouse interaction

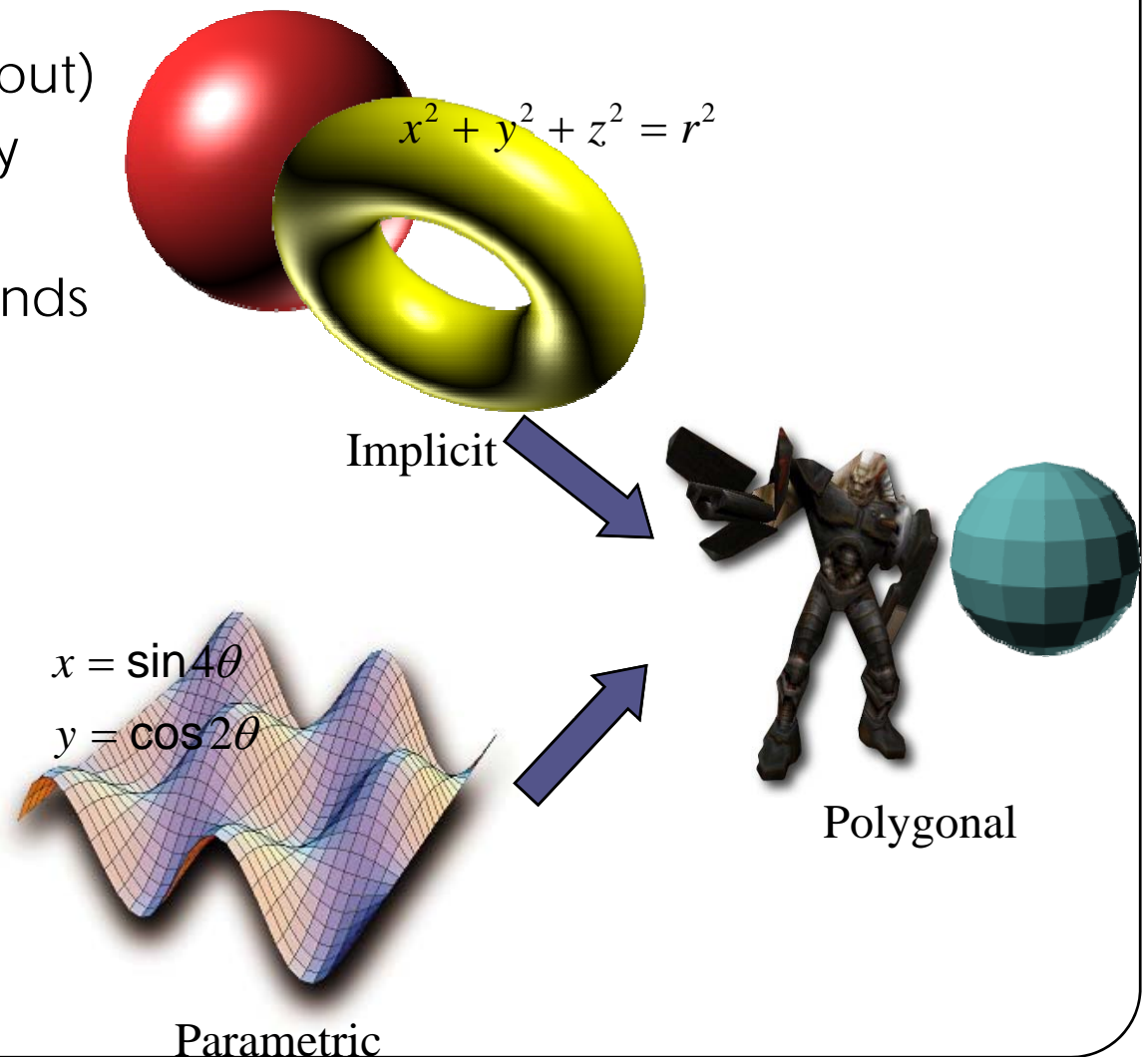
Graphics Pipeline Steps

1. Application
2. Command
3. Geometry
4. Rasterisation
5. Texturing
6. Fragment Operations
7. Display

(Akeley, Hanrahan 2001)

Application & Command

- Application
 - Handle events (User input)
 - Decide what to display
 - Generate primitives
 - Issue graphics commands
- Command
 - Buffer commands
 - Interpret commands
 - Perform conversions
 - Maintain graphics stat

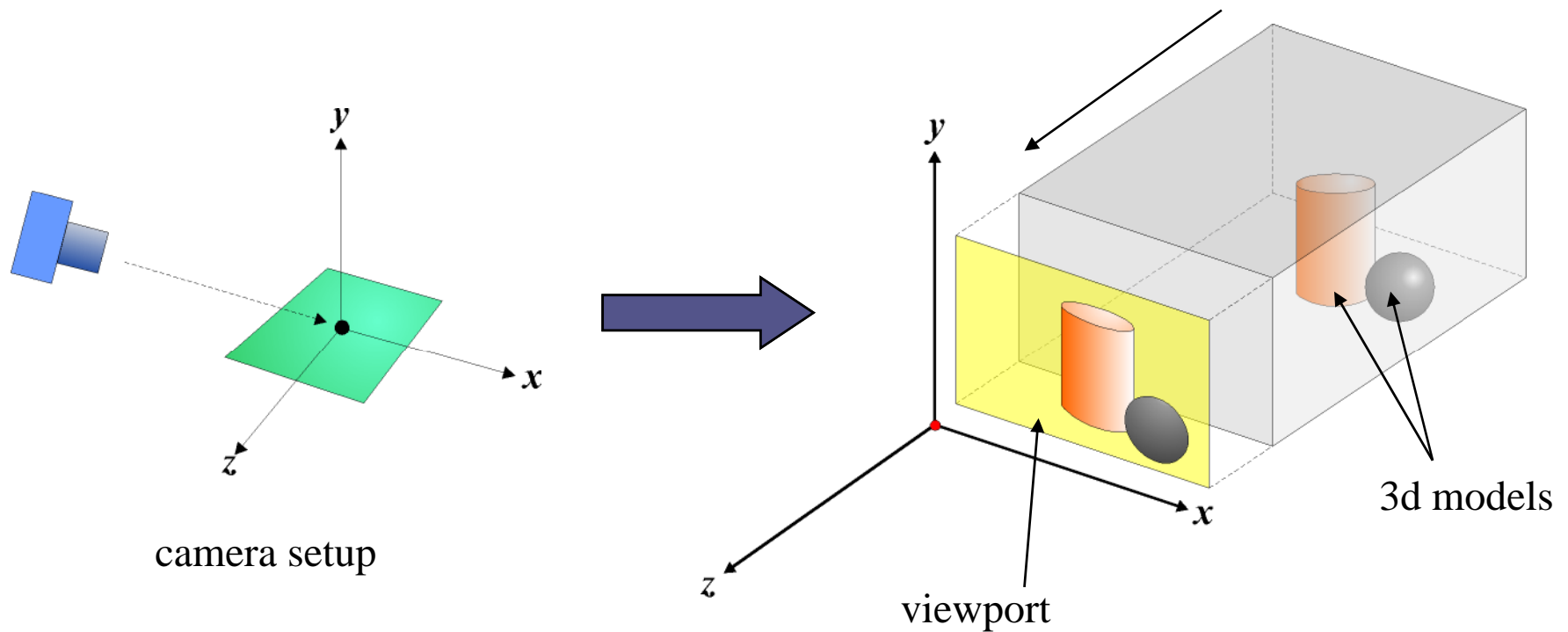


Geometry

- Evaluate primitives that need evaluation
- Transform and project geometry
 - Geometry shader
- Clip
- Cull
- Generate texture coordinates
- Generate lighting information
 - Vertex shader

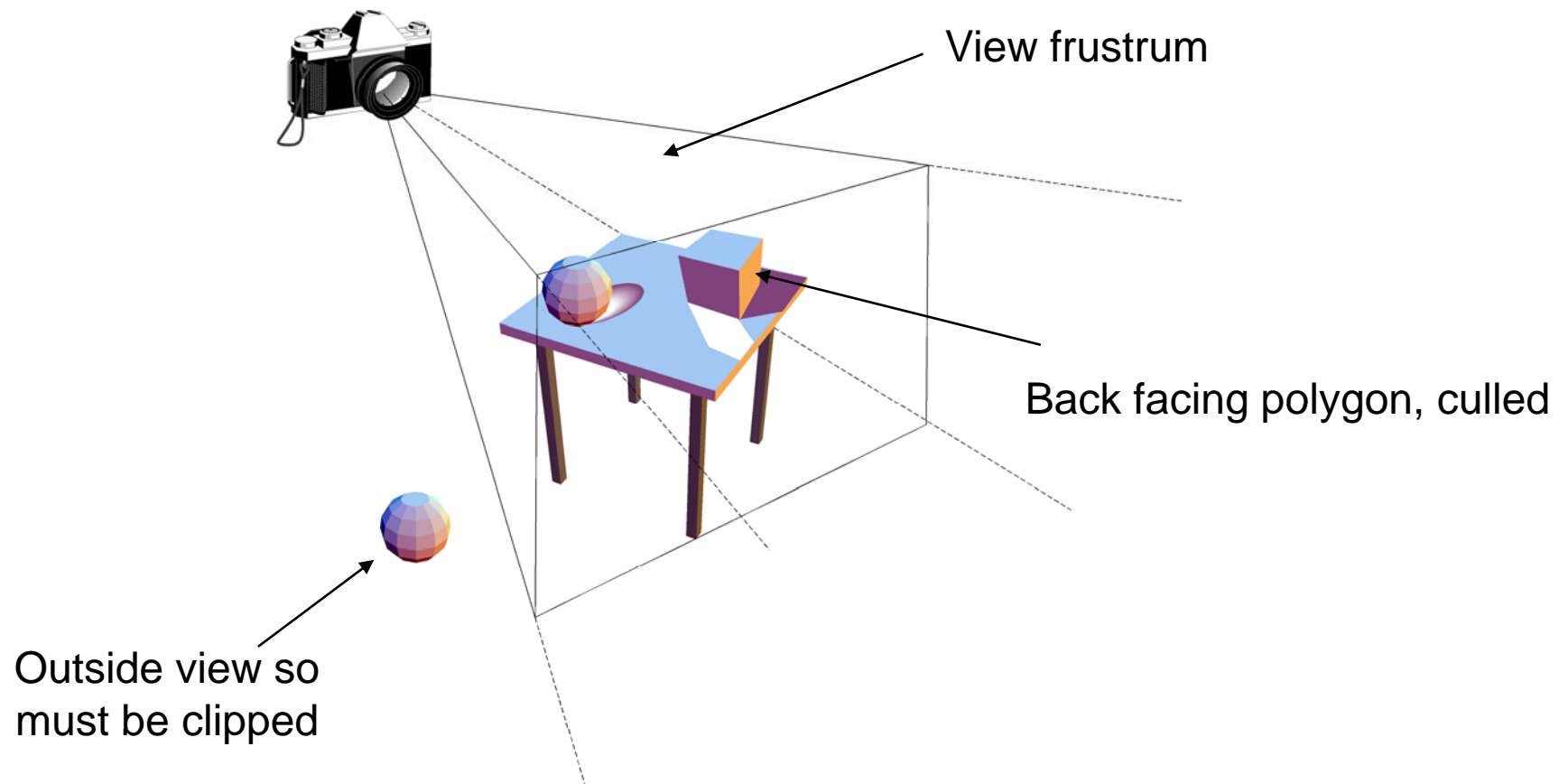
Geometry

Viewing & Projection



Geometry

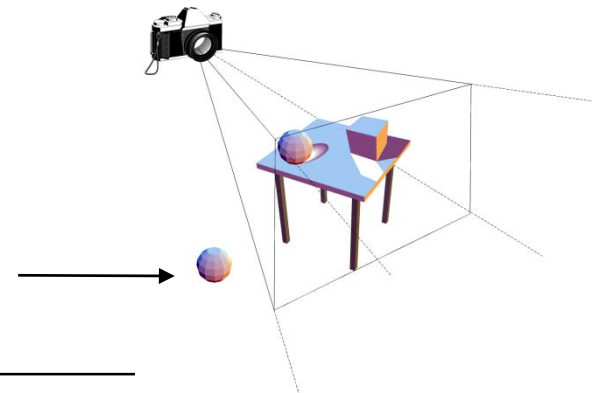
Clipping & Culling



Clipping

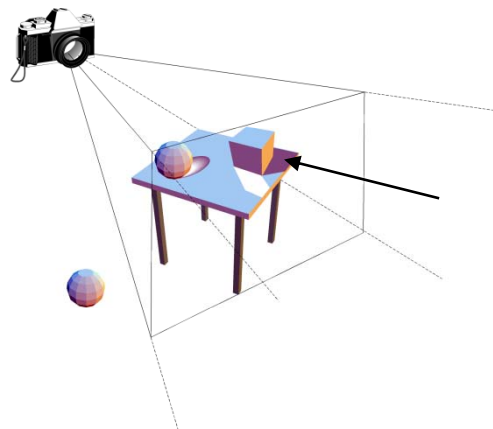
- The computer may have model, texture, and shader data for all objects in the scene in memory
- The virtual camera viewing the scene only “sees” the objects within the field of view
- The computer does not need to transform, texture, and shade the objects that are behind or on the sides of the camera
- A clipping algorithm skips these objects making rendering more efficient

Outside view so
must be clipped



Culling

- Polygons within the field of view are not drawn if they are occluded by other polygons
- Significant rendering time can be saved by doing a back-face culling pass before deciding which polygons to draw



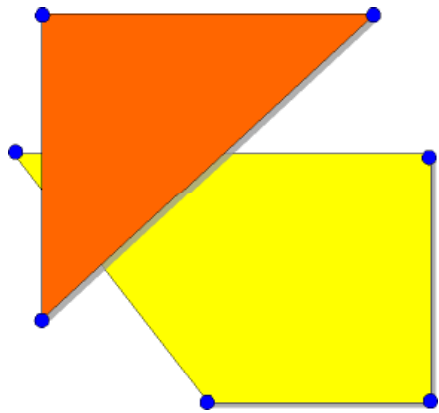
Back facing polygon, culled

Clipping/Culling Issues

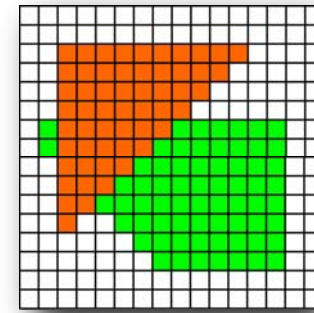
- A routine is needed to determine if the object is within the viewport or cut off at the borders
- If they are intersected by the viewport their shape needs to be adjusted
- Reflective surfaces (for example a mirror)
 - Can show objects that have been clipped
 - One solution is to let the clipper do a clipping and rendering pass from the point of view of the mirror and a clipping pass from the camera

Rasterisation

Screen (image) space
primitives (triangles)



Fragments



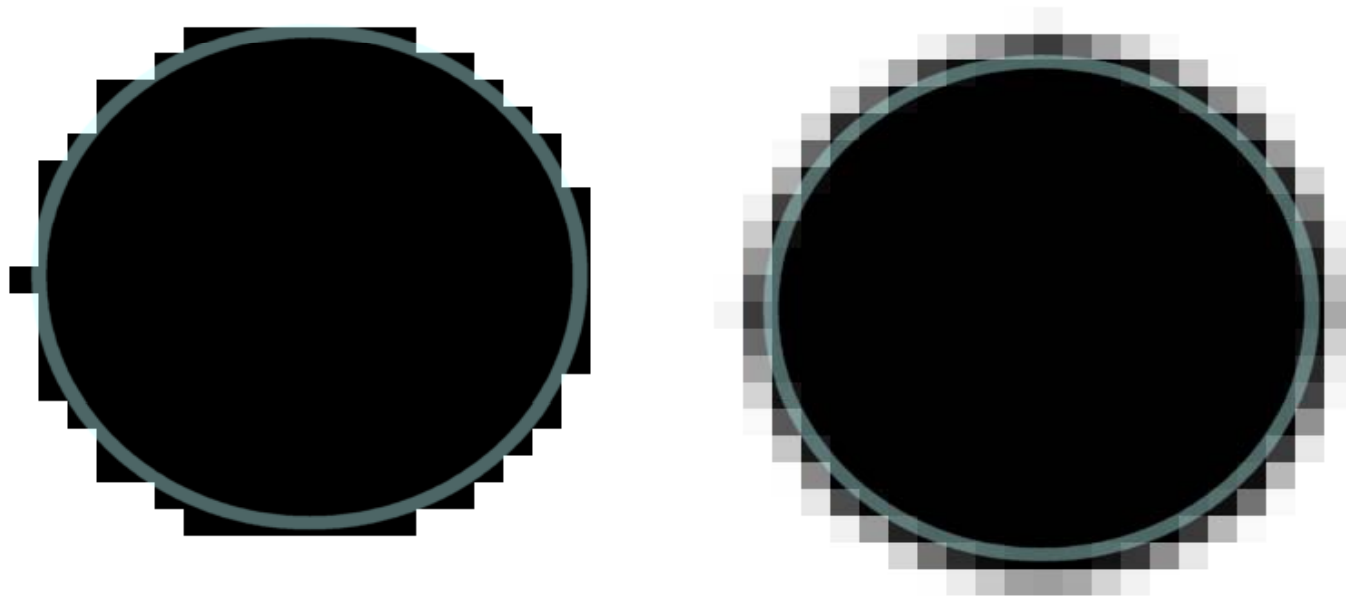
- Sampling
- Interpolation

Rasterisation

- Line drawing algorithms “render” a mathematical ideal of a line onto raster storage
- Similar techniques for curve-drawing and shape filling



Aliasing

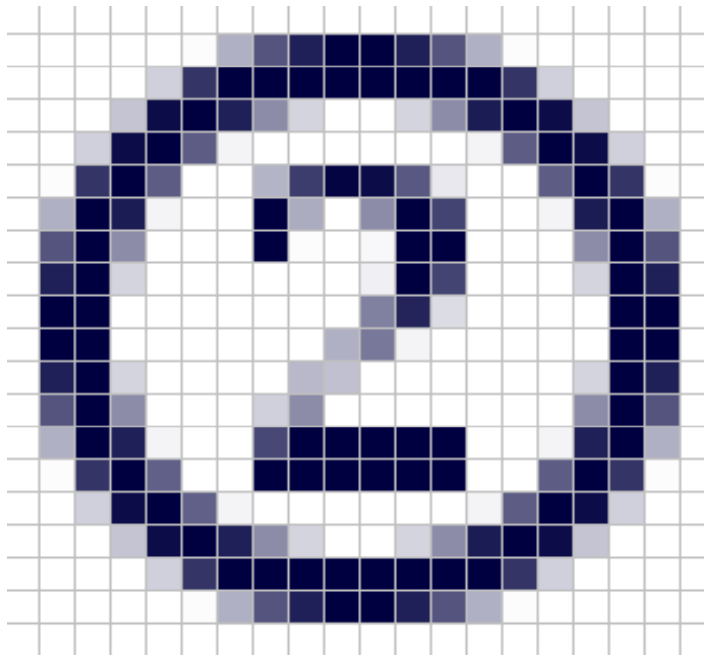


- *Anti-aliasing* is a technique for reducing the jagged edges (also known as jaggies) created when approximating smooth edges using pixels
- Anti-aliasing processes effectively blur edges to soften the boundaries
- Jaggies are usually the most visible, with near-horizontal or near-vertical lines

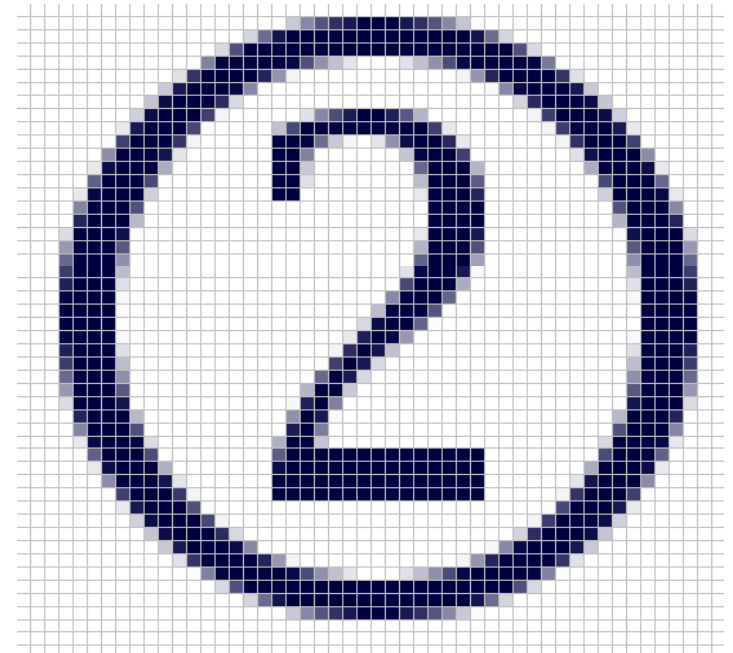
Resolution

- Rasterising objects effectively binds them to a certain raster level of detail.

②



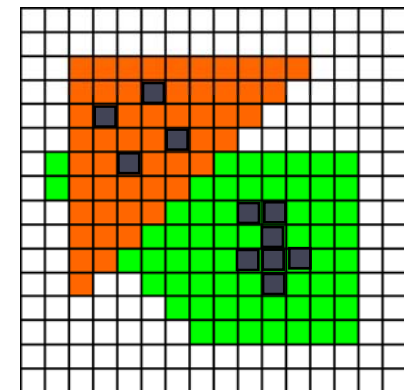
②



Texturing

- Applying texture images onto objects can make them look more realistic
- A texture is stored in memory
- Texture lookup
 - Texture transformation and projection
 - Address calculation
 - Texture filtering

**Geometry +
bitmap as
texture**



Fragment

- Combine textures
- Apply effects (e.g. fog)
- Clip in screen space (scissors)
- Do stenciling
- Test for depth (for hidden-surface removal)
- Blend and compose (using alpha = 0.0 complete transparency)
- Perform dithering

Display

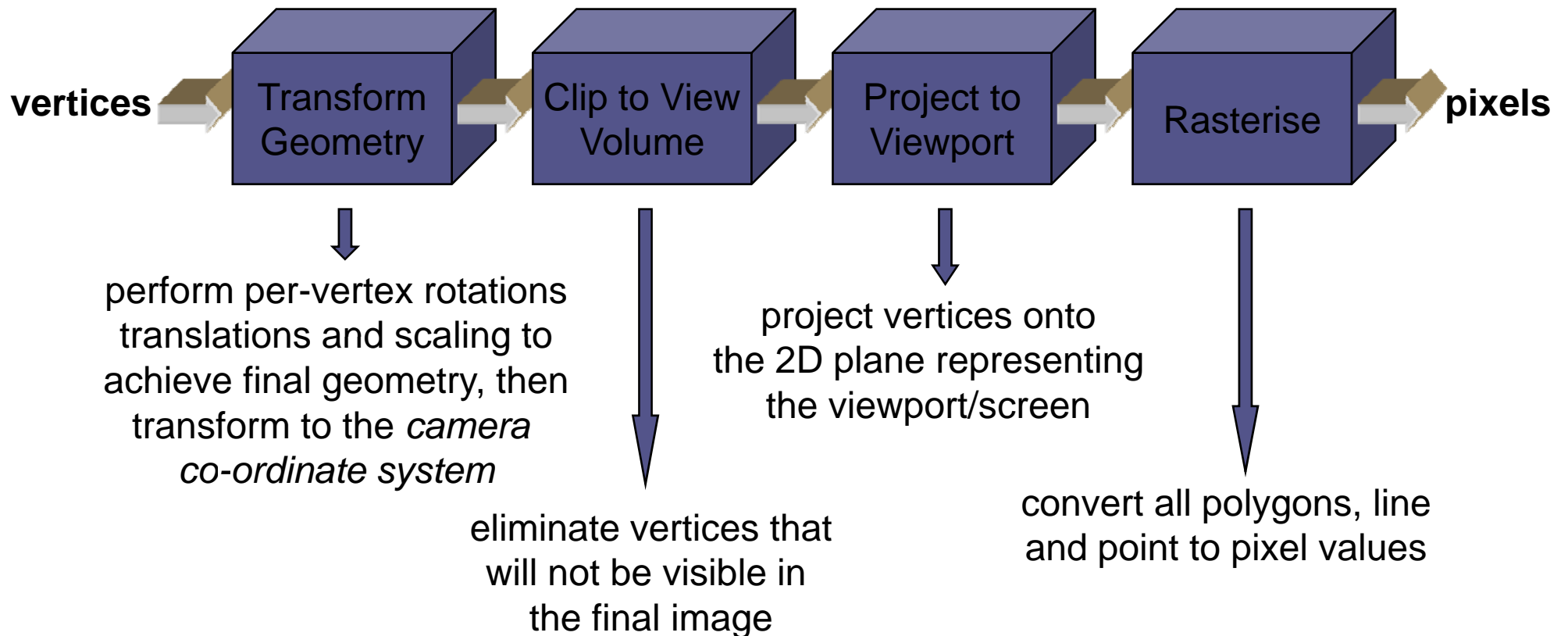
- Apply colour correction (gamma correction)
- Transform colours to voltages
- Send to display

Framebuffer pixels



Light

OpenGL Pipeline Overview



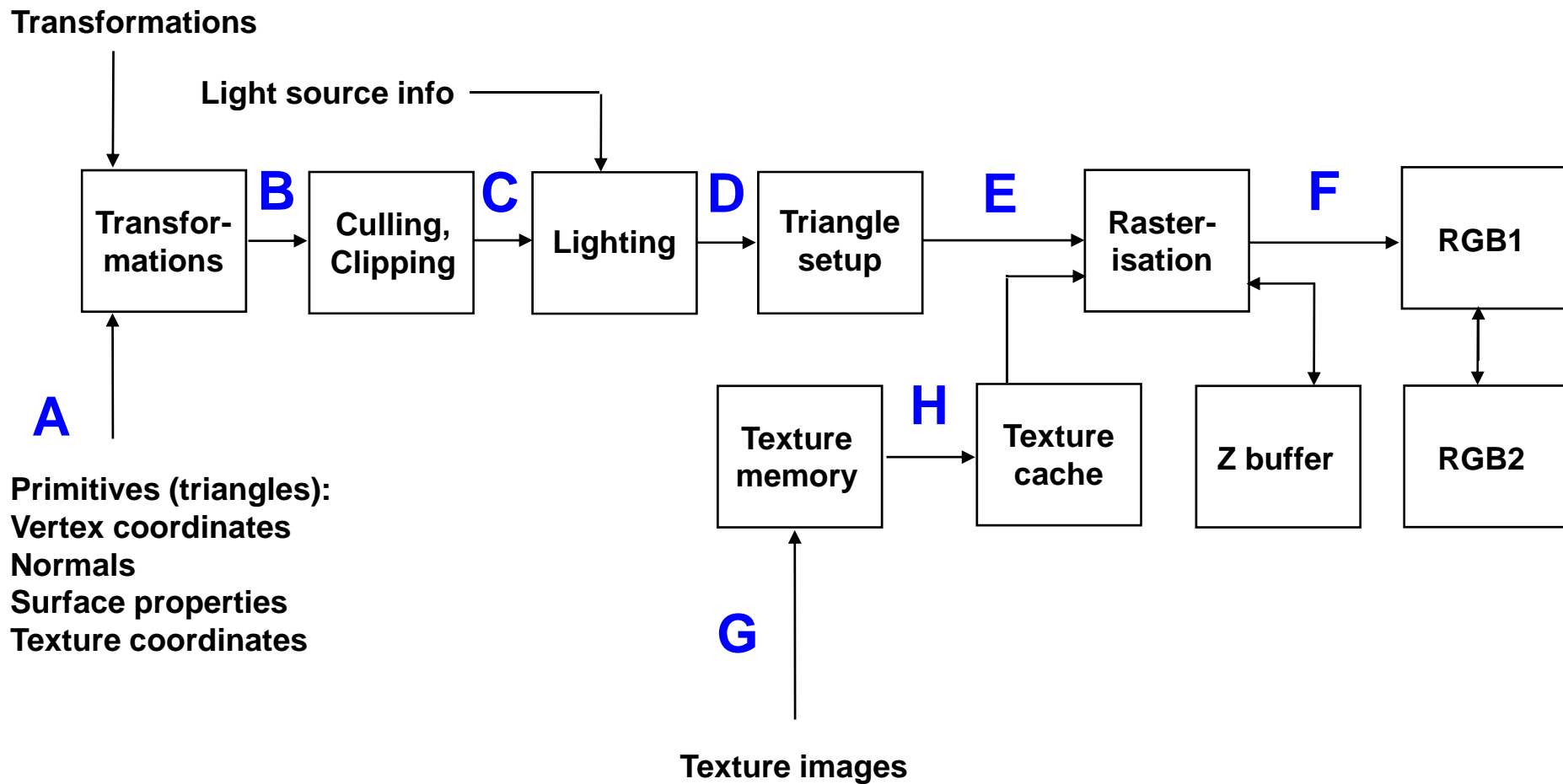
OpenGL Vertices

- OpenGL uses a 4 component vector to represent a vertex.
- Known as a homogenous coordinate system
- $z = 0$ in 2D space
- $w = 1$ usually

$$v = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

For further information on homogenous coordinate system, see Appendix G of the Red Book:
<http://fly.cc.fer.hr/~unreal/theredbook/appendixg.html>

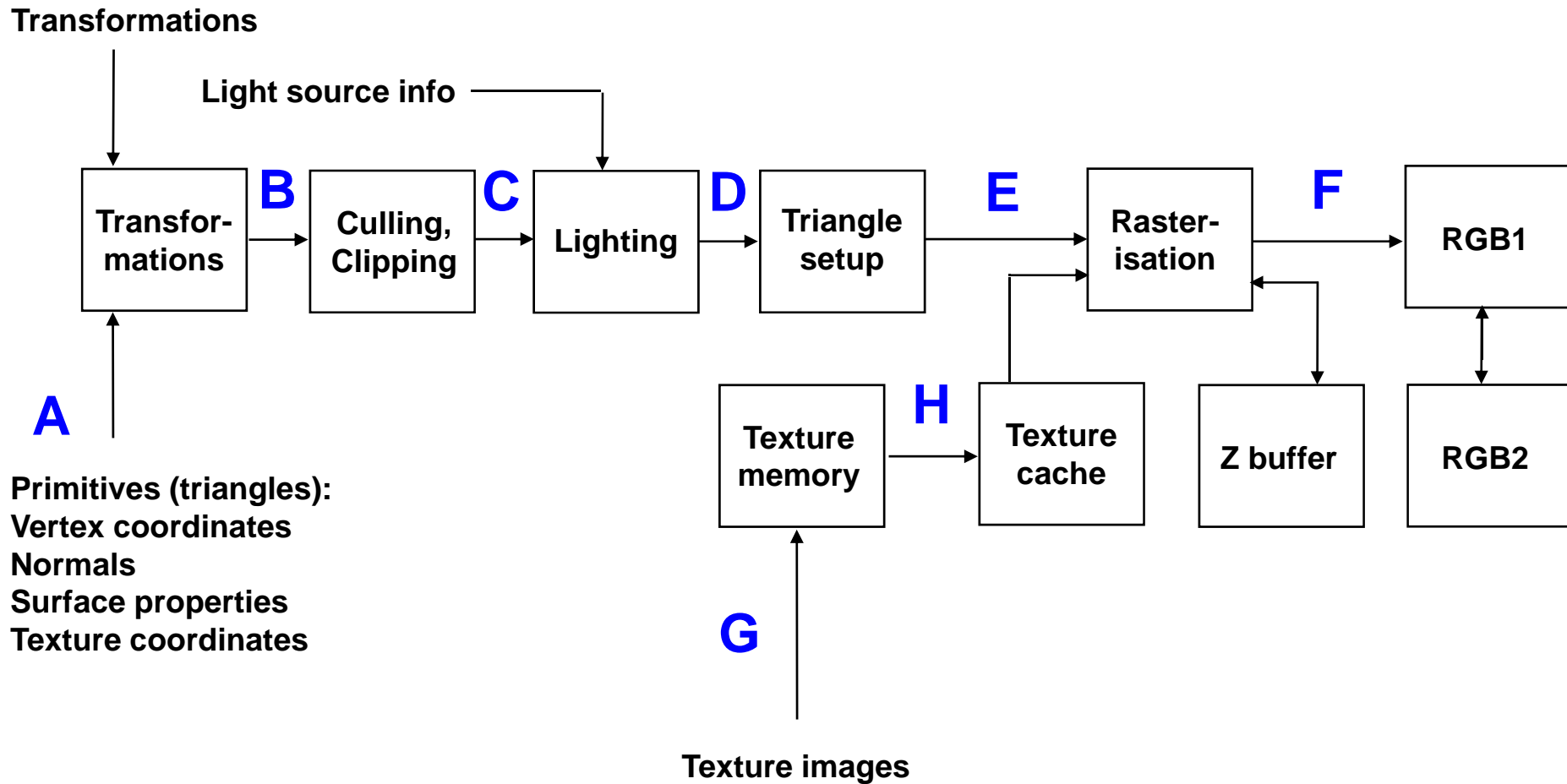
Graphics Pipeline



Graphics Pipeline

- **A**: The scene is described in terms of triangle primitives, light sources, and various transformations (object, perspective, viewport)
- Transformations are implemented as matrix multiplications in homogenous coordinates
- A last and important type of data is texture images
- Vertex coordinates and geometry data entering the pipeline at **A** are transformed along with their normals to a pixel-based window coordinate system suitable for rendering

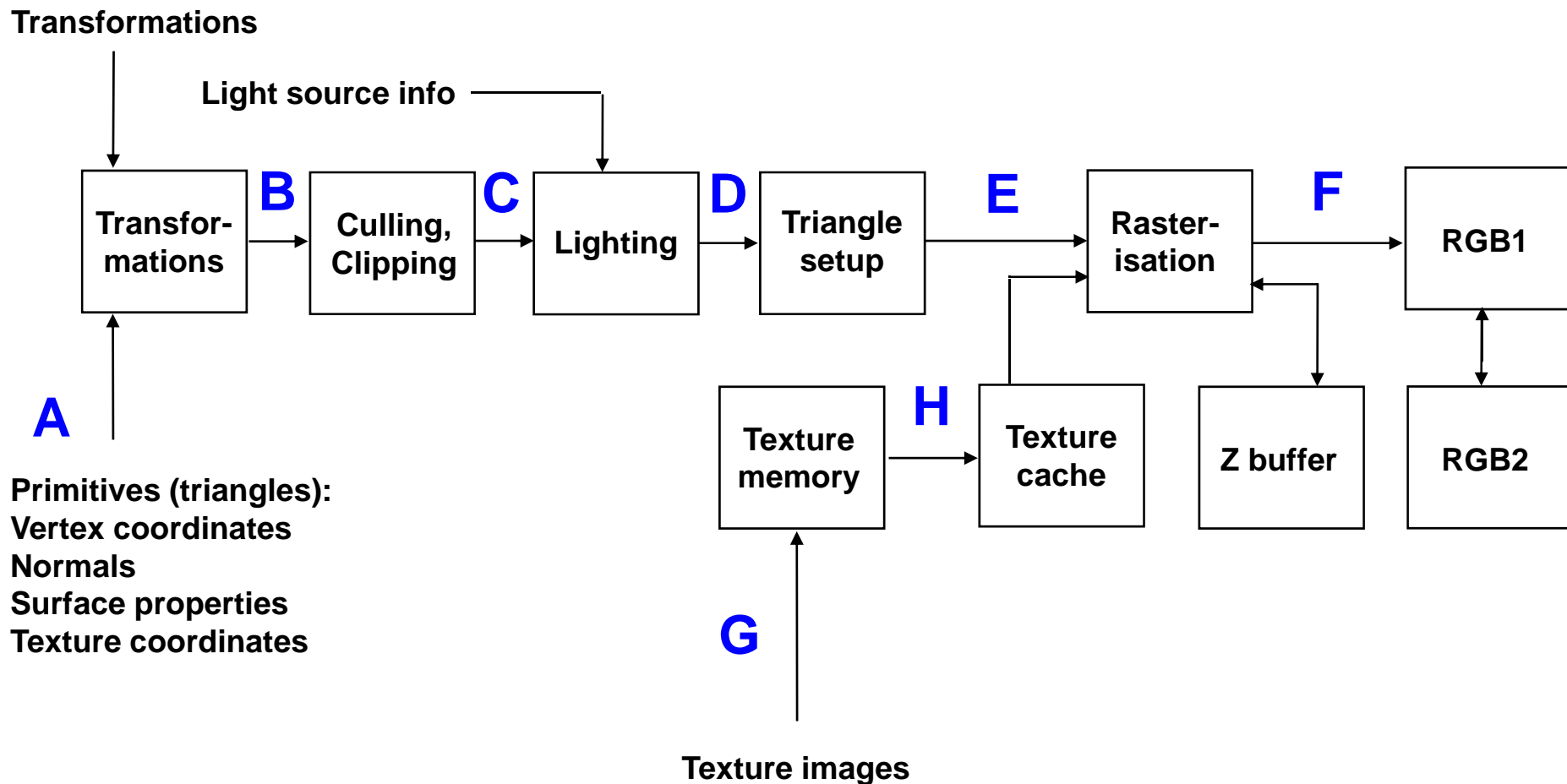
Graphics Pipeline



Graphics Pipeline

- **B**: The transformed geometry at **B** is then back face culled (if desired) and clipped against the viewport boundaries to obtain a modified set of geometric primitives **C** that are entirely within the viewport
- Per-vertex lighting calculations are performed

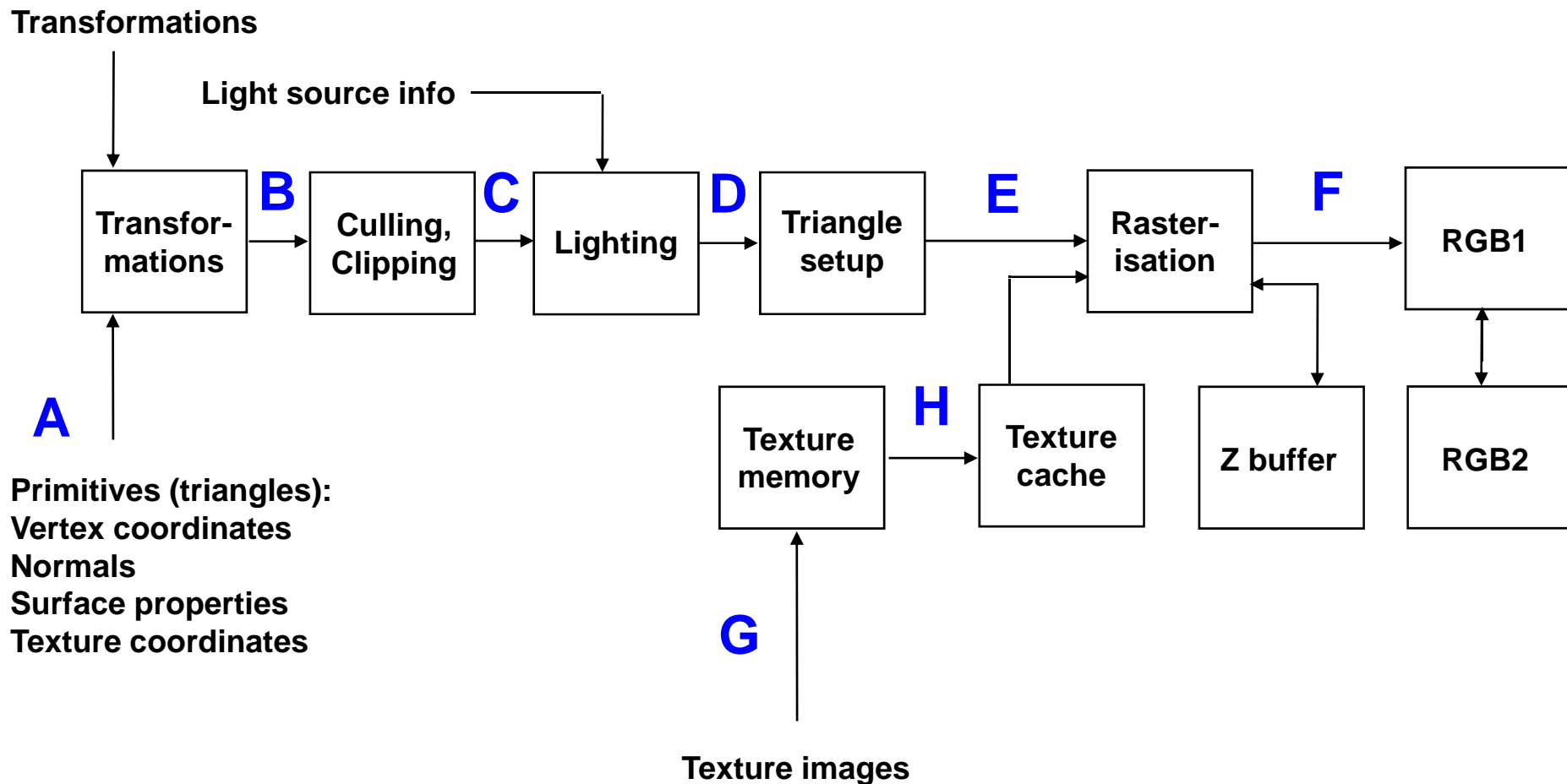
Graphics Pipeline



Graphics Pipeline

- **D**: The transformed, clipped and lit vertex data at **D** is then converted to spans
- Each triangle is split into one span per line of pixels in the output image
- All rendering is then performed through one-dimensional interpolation of values along the spans **E**
- This greatly simplifies the hardware required and speeds up the process considerably

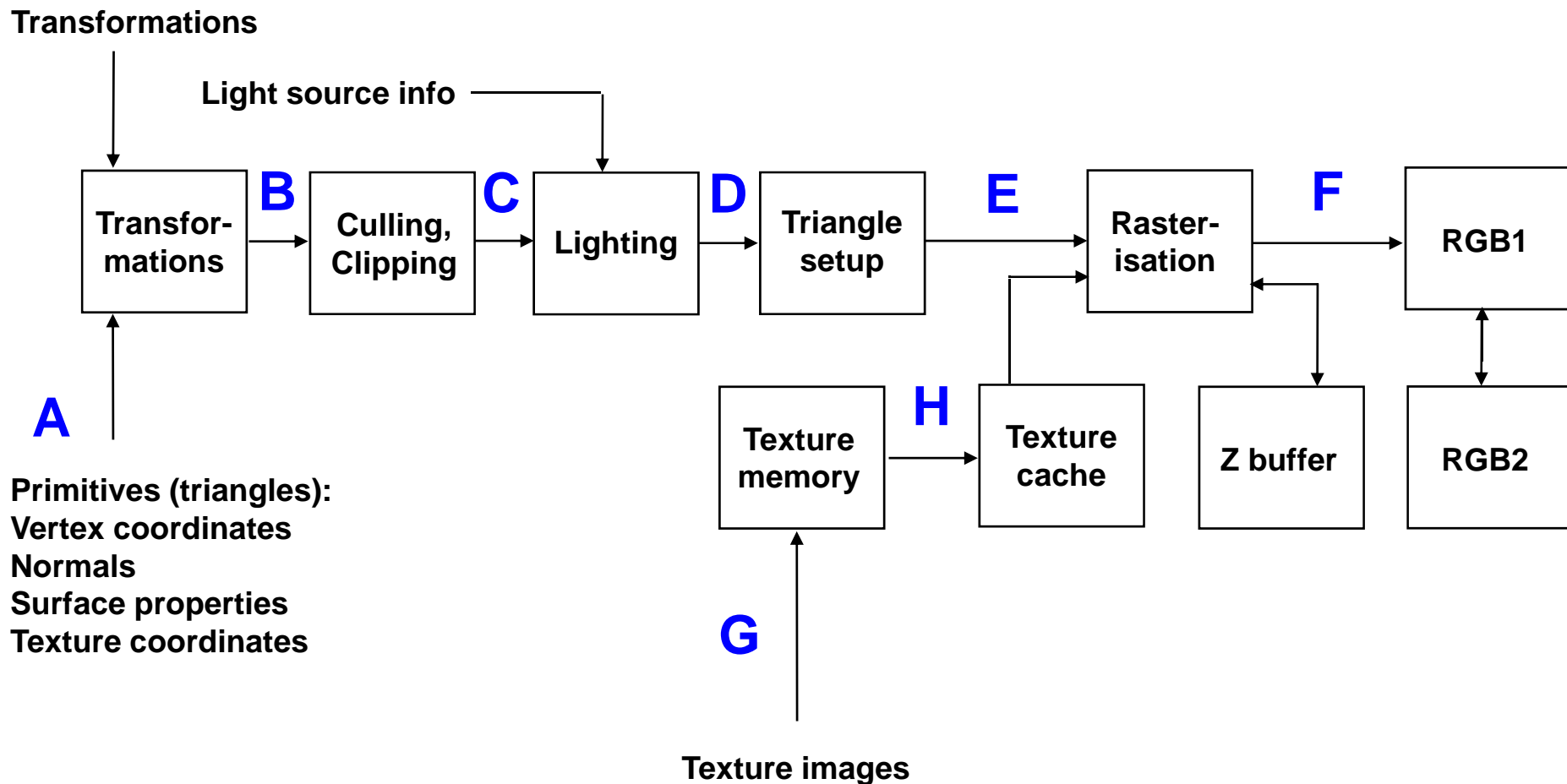
Graphics Pipeline



Graphics Pipeline

- **F**: The rasterised output F is written for each triangle in turn, not in pixel order from the top left corner in the output image
- This calls for a fast local memory for the image buffers, as pixels need to be accessed in a somewhat random order
- At least two buffers are required, one to display the latest fully rendered image and one to render the next image into

Graphics Pipeline



Graphics Pipeline

- The depth buffer is required only during rendering and can be shared between the two buffers
- **G**: Texture images are downloaded to a local texture memory for speed and efficiency reasons, and a small texture cache is used to render each span
- Interpolation hardware takes care of texture antialiasing through mipmapping and interpolation

OpenGL Conventions

- Conventions:
 - all function names begin with **gl**, **glu** or **glut**
 - **glBegin(...)**
 - **gluCylinder(...)**
 - **glutInitDisplayMode(...)**
 - constants begin with **GL_**, **GLU_**, or **GLUT_**
 - **GL_POLYGON**
 - Function names can encode parameter types, e.g. **glVertex***:
 - **glVertex2i(1, 3)**
 - **glVertex3f(1.0, 3.0, 2.5)**
 - **glVertex4fv(array_of_4_floats)**

The Drawing Process

- `ClearTheScreen() ;`
- `DrawTheScene() ;`
- `CompleteDrawing() ;`
- `SwapBuffers() ;`
- In animation there are usually two buffers. Drawing usually occurs on the background buffer. When it is complete, it is brought to the front (swapped). This gives a smooth animation without the viewer seeing the actual drawing taking place. Only the final image is viewed.
- The technique to swap the buffers will depend on which windowing library you are using with OpenGL.

Clearing the Window

- `glClearColor(0.0, 0.0, 0.0, 0.0);`
- `glClear(GL_COLOR_BUFFER_BIT);`
- Typically you will clear the colour and depth buffers.
- `glClearColor(0.0, 0.0, 0.0, 0.0);`
- `glClearDepth(0.0);`
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
- You can also clear the accumulation and stencil buffers.
 - `GL_ACCUM_BUFFER_BIT` and `GL_STENCIL_BUFFER_BIT`

Specifying a Colour

- Colour is specified in (R,G,B,A) form [Red, Green, Blue, Alpha], with each value being in the range of 0.0 to 1.0.
- There are many variants of the glColor command

- `glColor4f(red, green, blue, alpha);`

- `glColor3f(red, green, blue);`

- `glColor3f(0.0, 0.0, 0.0); /* Black */`

- `glColor3f(1.0, 0.0, 0.0); /* Red */`

- `glColor3f(0.0, 1.0, 0.0); /* Green */`

- `glColor3f(1.0, 1.0, 0.0); /* Yellow */`

- `glColor3f(1.0, 0.0, 1.0); /* Magenta */`

- `glColor3f(1.0, 1.0, 1.0); /* White */`

Complete Drawing the Scene

- Need to tell OpenGL you have finished drawing your scene.

- `glFinish();`

or

- `glFlush();`

- For more information see Chapter of the Red Book:

- <http://fly.cc.fer.hr/~unreal/theredbook/chapter02.html>

Drawing in OpenGL

- Use `glBegin()` to start drawing, and `glEnd()` to stop.
- `glBegin()` can draw in many different styles.
- `glVertex3f(x,y,z)` specifies a point in 3D space.

```
glBegin(GL_POLYGON);  
glVertex3f(0.0, 0.0, 0.0);  
glVertex3f(0.0, 3.0, 1.0);  
glVertex3f(3.0, 3.0, 3.0);  
glVertex3f(4.0, 1.5, 1.0);  
glVertex3f(3.0, 0.0, 0.0);  
glEnd();
```

Drawing in OpenGL

- To create a red polygon with 4 vertices:

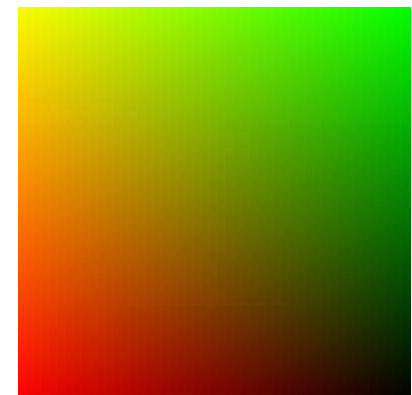
```
glColor3f(1.0, 0.0, 0.0);  
glBegin(GL_POLYGON);  
    glVertex3f(0.0, 0.0, 3.0);  
    glVertex3f(1.0, 0.0, 3.0);  
    glVertex3f(1.0, 1.0, 3.0);  
    glVertex3f(0.0, 1.0, 3.0);  
glEnd();
```

- **glBegin** defines a *geometric primitive*:
 - **GL_POINTS**, **GL_LINES**, **GL_LINE_LOOP**, **GL_TRIANGLES**, **GL_QUADS**, **GL_POLYGON**...
- All vertices are 3D and defined using **glVertex***

Mixing Geometry with Colour

- Specifying vertices can be mixed with colour and other commands for interesting drawing results.
- We can use *per-vertex* information.
- To create the RG colour square:

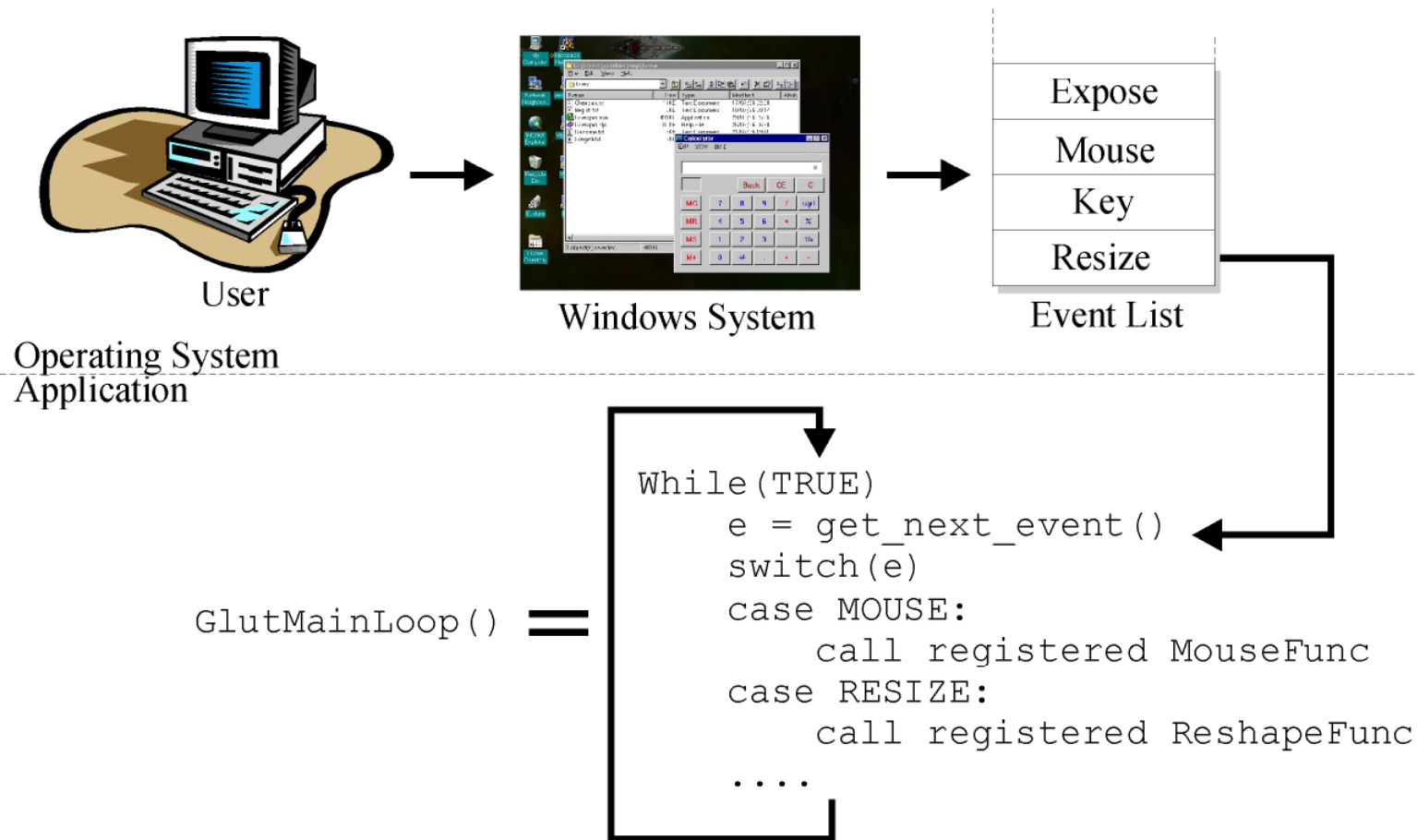
```
glShadeModel(GL_SMOOTH);  
glBegin(GL_POLYGON);  
    glColor3f(1.0, 0.0, 0.0); // Red  
    glVertex3f(0.0, 0.0, 3.0);  
    glColor3f(0.0, 0.0, 0.0); // Black  
    glVertex3f(1.0, 0.0, 3.0);  
    glColor3f(0.0, 1.0, 0.0); // Green  
    glVertex3f(1.0, 1.0, 3.0);  
    glColor3f(1.0, 1.0, 0.0); // Yellow  
    glVertex3f(0.0, 1.0, 3.0);  
glEnd();
```



OpenGL GLUT Event Loop

- Interaction with the user is handled through an *event loop*.
- Application registers *handlers* (or *callbacks*) to be associated with particular events:
 - mouse button, mouse motion, timer, resize, redraw
- GLUT provides a wrapper on the X-Windows or Win32 core event loop.
- X-Windows or Win32 manages event creation and passing, GLUT uses them to catch events and then invokes the appropriate callback.
- GLUT is more general than X or Win32 etc.
 - ⇒ more portable: user interface code need not be changed.
 - ⇒ less powerful: implements a common subset

OpenGL GLUT Event Loop



OpenGL GLUT Event Loop

- To add handlers for events we call a callback registering function, e.g:

```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
```

- Takes a function (the required callback) as a parameter.
- Handlers must conform to the specification defined.
- Example:

```
void key_handler(unsigned char key, int x, int y);  
glutKeyboardFunc(key_handler);
```

- In this case, **key** is the ascii code of the key hit and **(x,y)** is the mouse position within the window when the key was hit.
- The callback function is *automatically* called when a key is hit.

OpenGL GLUT Initialisation

Pass command line arguments
to GLUT system

RGB Colour, depth testing
and double buffering

```
glutInit(&argc, argv);  
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);  
glutCreateWindow("RGSquare Application");  
glutReshapeWindow(400, 400);
```

Window title

Request for window size
(not necessarily accepted)

OpenGL GLUT Callback Registration

```
void reshape(int w, int h) {...}
void keyhit(unsigned char c, int x, int y) {...}
void idle(void) {...}
void motion(int x, int y) {...}
void mouse(int button, int state, int x, int y) {...}
void visibility(int state) {...}
void timer(int value) {...}
```

GLUT_LEFT_BUTTON
GLUT_MIDDLE_BUTTON
GLUT_RIGHT_BUTTON

GLUT_UP OR GLUT_DOWN

timer ID

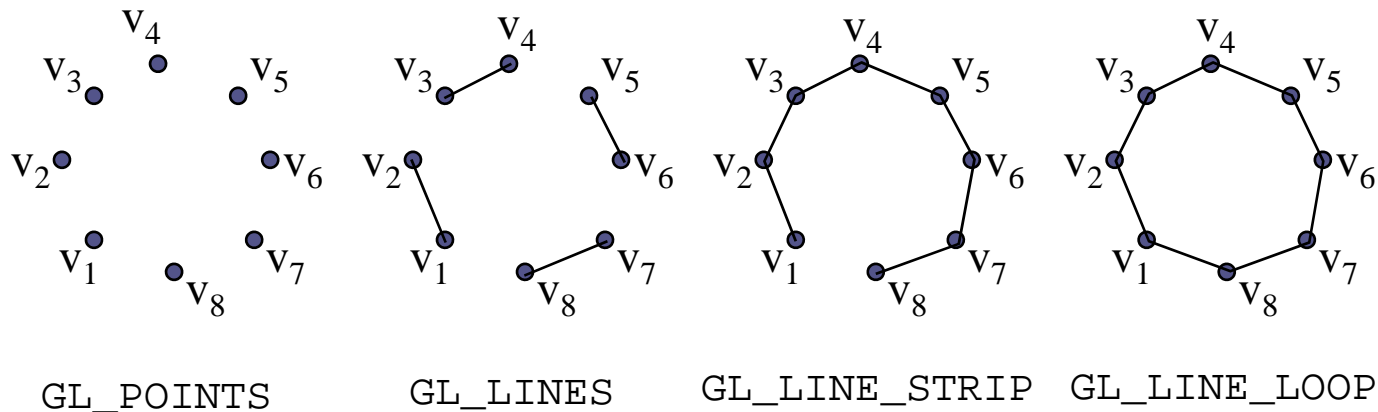
GLUT_NOT_VISIBLE OR GLUT_VISIBLE

<code>glutReshapeFunc(reshape);</code>	→	window resized
<code>glutKeyboardFunc(keyhit);</code>	→	key hit
<code>glutIdleFunc(idle);</code>	→	system idle
<code>glutDisplayFunc(draw);</code>	→	window exposed
<code>glutMotionFunc(motion);</code>	→	mouse moved
<code>glutMouseFunc(mouse);</code>	→	mouse button hit
<code>glutVisibilityFunc(visibility);</code>	→	window (de)iconified
<code>glutTimerFunc(timer);</code>	→	timer elapsed

`glutMainLoop();` ← Begin infinite event loop

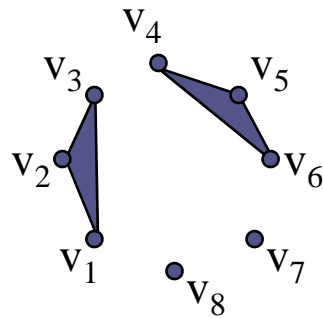
OpenGL Primitives

- All geometric objects in OpenGL are created from a set of basic *primitives*.
- Certain primitives are provided to allow optimisation of geometry for improved rendering speed.
- Line based primitives:

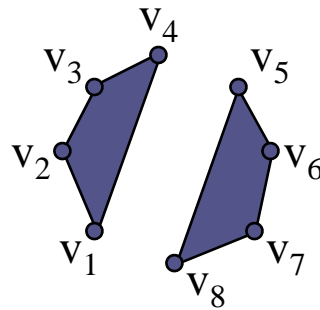


OpenGL® Primitives

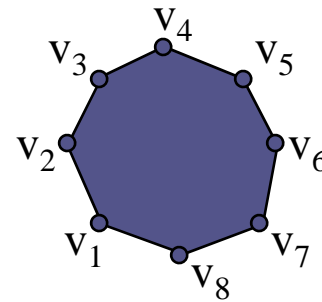
- Polygon primitives



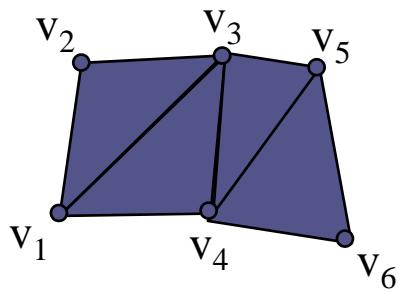
GL_TRIANGLES



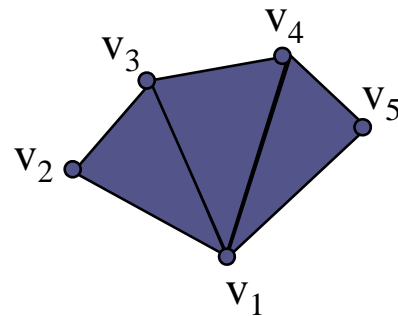
GL_QUADS



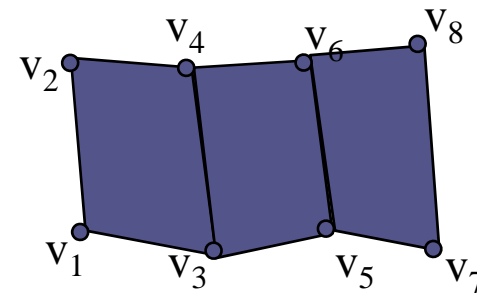
GL_POLYGON



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUAD_STRIP

OpenGL Objects: GLU

- GLU provides functionality for the creation of quadric surfaces
 - spheres, cones, cylinders, disks
- A quadric surface is defined by the following implicit equation:

$$ax^2 + by^2 + cz^2 + dxy + eyz + fxz + gx + hy + iz + j = 0$$

$$\vec{x}^T \mathbf{Q} \vec{x} + j = 0 \text{ where } \vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{ and } \mathbf{Q} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

```
GLUquadricObj* gluNewQuadric(void);
```

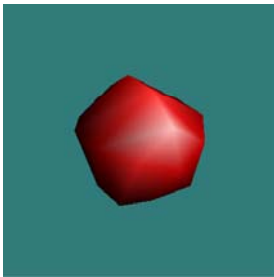
← **use to initialise a quadric**

```
gluDeleteQuadric(GLUquadricObj *obj);
```

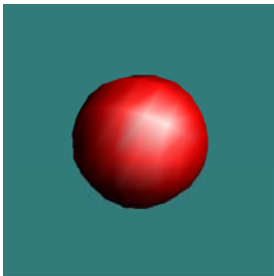
← **delete when finished**

OpenGL Objects: GLU Spheres

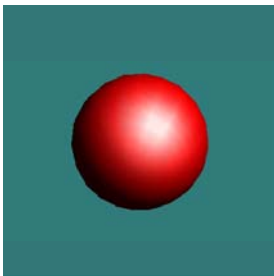
```
void gluSphere(GLUquadricObj *obj, double radius, int slices, int stacks);
```



```
gluSphere(obj, 1.0, 5, 5);
```



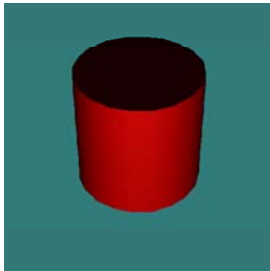
```
gluSphere(obj, 1.0, 10, 10);
```



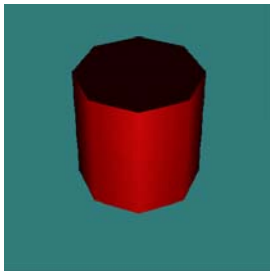
```
gluSphere(obj, 1.0, 20, 20);
```

Other GLU Quadrics

```
void gluCylinder(GLUquadricObj *obj, double base_radius, double top_radius,  
double height, int slices, int stacks);
```



```
gluCylinder(obj, 1.0, 1.0, 2.0, 20, 8);
```



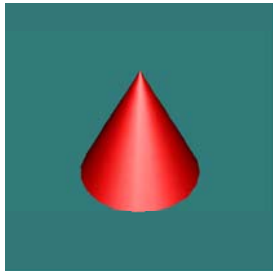
```
gluCylinder(obj, 1.0, 1.0, 2.0, 8, 8);
```



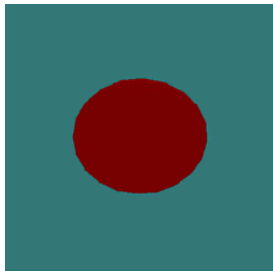
```
gluCylinder(obj, 1.0, 0.3, 2.0, 20, 8);
```

Other GLU Quadrics

```
void gluDisk(GLUquadricObj *obj, double inner_radius, double outer_radius,  
            int slices, int rings);
```



```
gluCylinder(obj, 1.0, 0.0, 2.0, 20, 8);
```



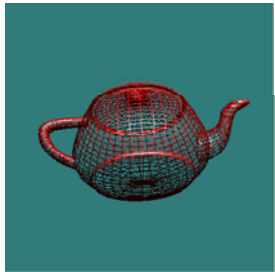
```
gluDisk(obj, 0.0, 2.0, 10, 3);
```



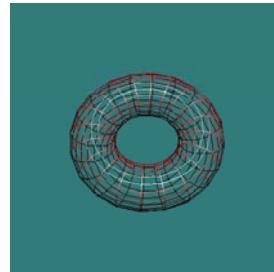
```
gluDisk(obj, 0.5, 2.0, 10, 3);
```

OpenGL® Objects: GLUT

```
void glutSolidTorus(double inner_radius, double outer_radius,  
int nsides, int rings);
```



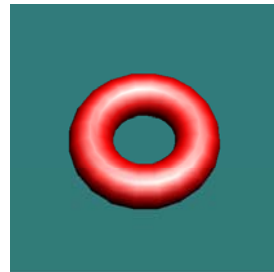
```
glutWireTeapot(1.0)
```



```
glutWireTorus(0.3, 1.5,  
20, 20);
```



```
glutSolidTeapot(1.0)
```



```
glutSolidTorus(0.3, 1.5,  
20, 20);
```

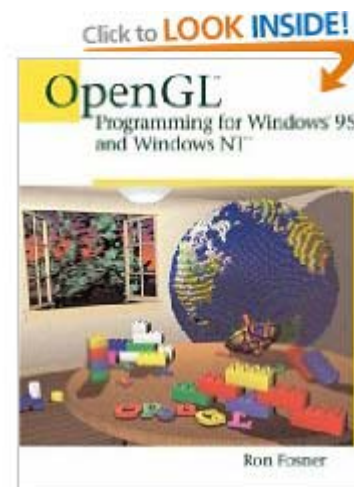
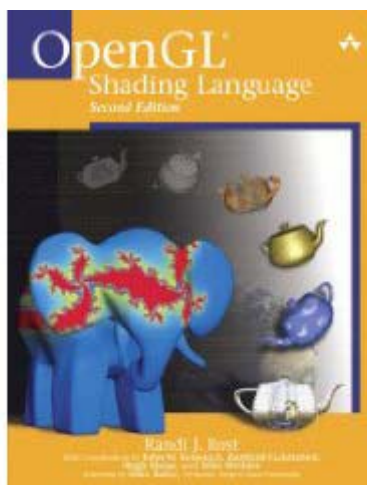
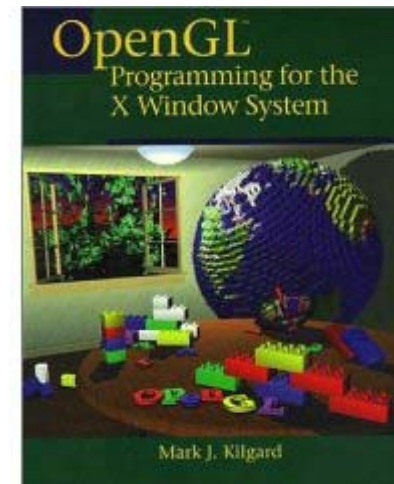
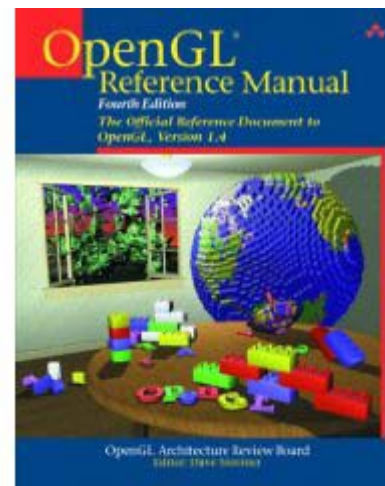
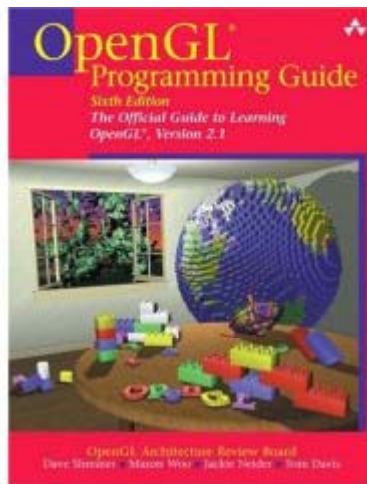


```
glutSolidDodecahedron()
```

size



OpenGL Books



Resources

- OpenGL Home Page
 - <http://www.opengl.org>
- OpenGL Tutors
 - <http://www.xmission.com/~nate/tutors.html>
- NeHe Tutorials
 - <http://nehe.gamedev.net>
- OpenGL Red Book (Programming Guide)
 - <http://www.glprogramming.com/red/>
- OpenGL Blue Book (Reference Manual)
 - <http://www.glprogramming.com/blue/>
- OpenGL video tutorials on various topics
 - <http://www.videotutorialsrock.com/>

Recommended Material

- Read Chapters 1-6 of OpenGL Red Book
- Familiarise yourself with OpenGL Blue Book
- Play with OpenGL Tutors
- Learn about GLUT
- Do NeHe Tutorial Lessons 1-5 (with GLUT)
- Have a look at the video tutorials (especially Part 1: The Basics and later Part 2: Topics in 3D Programming)