OpenGL
I –GLUT & Fixed Function Pipeline

Stefan BORNHOFEN

What is OpenGL?

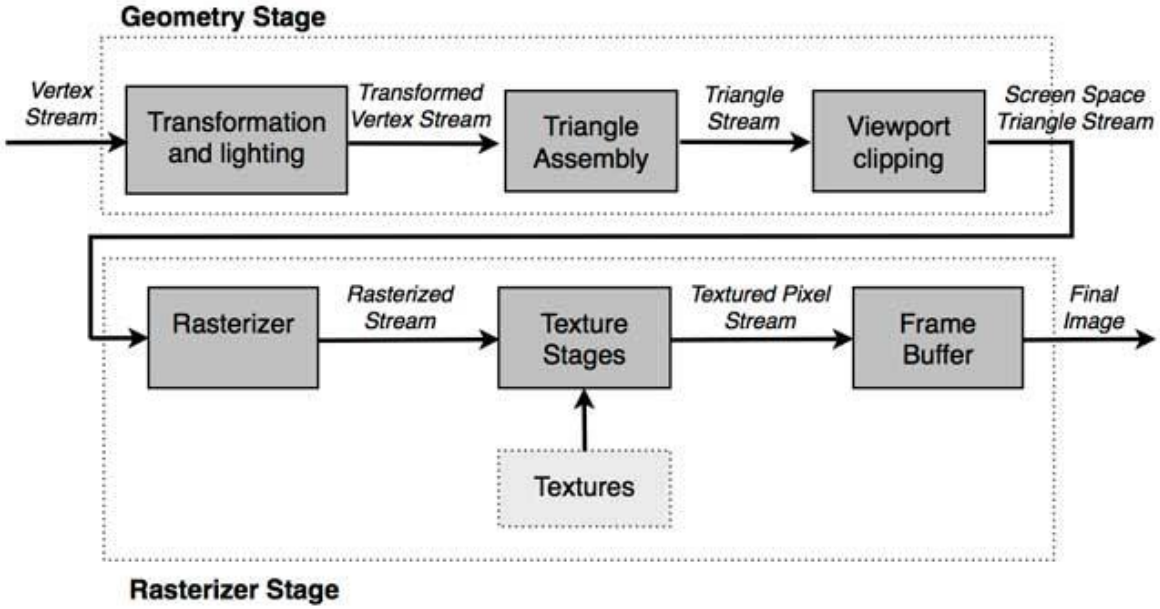
- API for 2D and 3D graphics
- First release in 1992, maintained by the Khronos group
- Bindings to many programming languages
- Interacts with GPU (hardware-accelerated rendering)
- Operating system independent

Competitors

- Direct3D: a proprietary API by Microsoft that provides equivalent functionalities for use on the Windows platform
- Vulkan: “Next generation OpenGL" released in 2016 by the Khronos group. Intended to provide a variety of advantages over OpenGL (unifying OpenGL and OpenGL ES, lower overhead, better GPU control, lower CPU usage)

Fixed-Function Pipeline (OpenGL 1.x)

- In the early days, OpenGL and DirectX had a fixed-function pipeline.
- The programmer only specified basic data: geometry description (vertices), textures, the position and orientation of the geometries, the position and orientation of the camera, lights and some more parameters
- No control, complex effects impossible to implement
- Nowadays, OpenGL and DirectX allow most steps in the pipeline to be programmable through the use of **shaders.**



OpenGL as a state machine

- Put OpenGL into states (modes).
  - Projection and viewing matrix
  - Color and material properties
  - Lights
  - Line and polygon drawing modes
  - …
- State variables can be set and queried.
- They remain unchanged until the next change.

Related Libraries

- **GLU (OpenGL Utility Library)**
  - Offers some higher level operations
  - More primitives: “quadrics” (sphere, cylinder, etc.)
  - NURBS
  - Concave polygon tessellators
  - A simpler viewing mechanism (“glulookat”)
- **GLUT (OpenGL Utility Toolkit)**
  - Portable windowing API for fast prototyping
  - Handles window creation and OS system calls (timer, mouse, keyboard, etc.)
  - Very limited GUI (no dialog boxes, menu bar, etc.)
  - If you need more, you can use GLUI

GLUT Basics

- Initialize GLUT and open window
- Initialize OpenGL state
- Register callback functions
  - render
  - keyboard
  - mouse
  - etc.
- Enter event processing loop

Let’s go! Install GLUT

Follow the GLUT part of the document

Setting up OpenGL, GLUT & GLEW for Visual C++ in Windows

```
#include <GL/freeglut.h>
#include <GL/glut.h>
#include <GL/gl.h>

#define TITLE "Hello OpenGL!"
int SCREEN_X = 1024;
int SCREEN_Y = 768;

void main(int argc, char** argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE);
	glutInitWindowSize(SCREEN_X, SCREEN_Y);
	glutCreateWindow(TITLE);
	glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,
		GLUT_ACTION_GLUTMAINLOOP_RETURNS);
	init();
	glutDisplayFunc(displayfunc);
	glutMainLoop();
	cleanup();
}
```

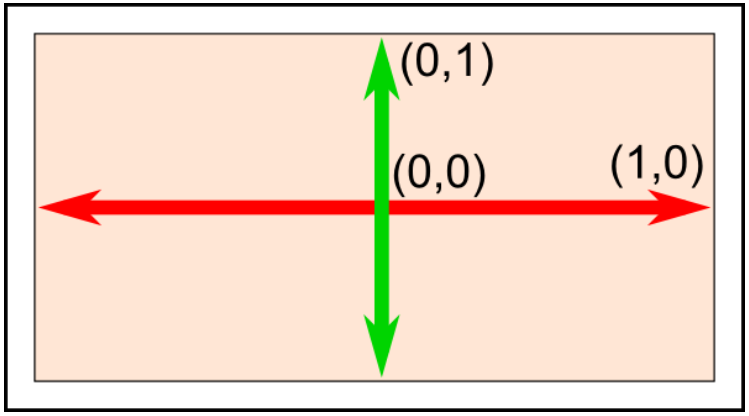
```
void init()
{
  glClearColor (0.0, 0.0, 0.0, 1.0);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -2.0, 2.0);

  glViewport(0,0,SCREEN_X,SCREEN_Y);
}
```

```
void displayfunc()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glutSwapBuffers();
}
```

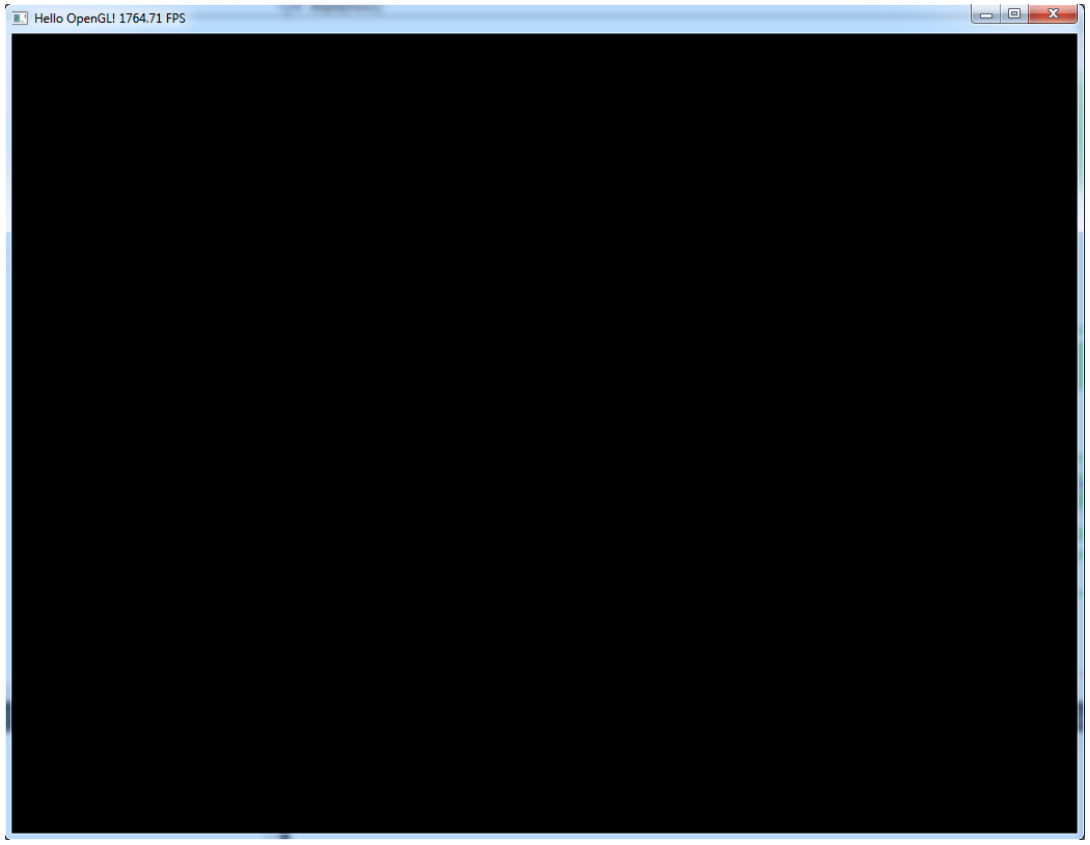
```
void cleanup()
{
  // free allocated memory
}
```



The OpenGL coordinate system is different from the window system

Exercise

Create a black GLUT window



Rendering method

- Callback function where all our drawing is done
- Every GLUT program must have a display callback

```
void displayfunc()
{
   glClear(GL_COLOR_BUFFER_BIT);

   glBegin(...); // immediate mode
     ...
     ...
     ...
   glEnd();

   glutSwapBuffers();
}
```

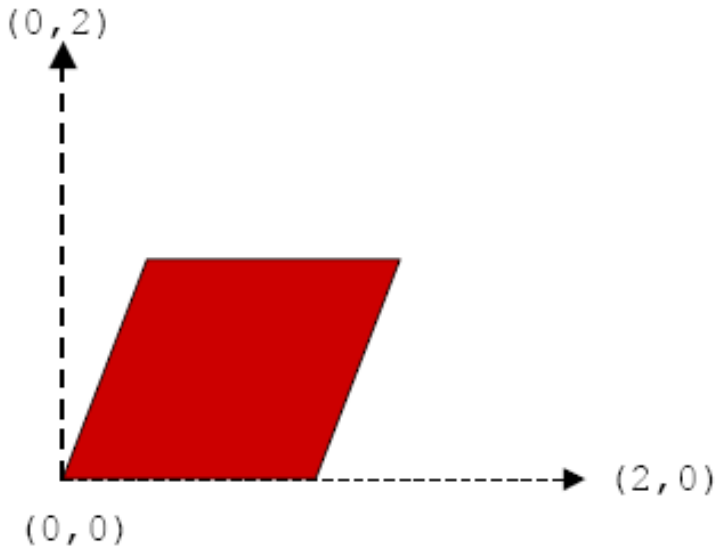
Immediate mode

Primitives are specified using

```
glBegin(primType);
...
glEnd();
```

Example

```
void drawParallelogram()
{
   glBegin(GL_QUADS);
     glColor3f(1.0,0.0,0.0);
     glVertex2f(0.0, 0.0);
     glVertex2f(1.0, 0.0);
     glVertex2f(1.5, 1.118);
     glVertex2f(0.5, 1.118);
   glEnd();
}
```

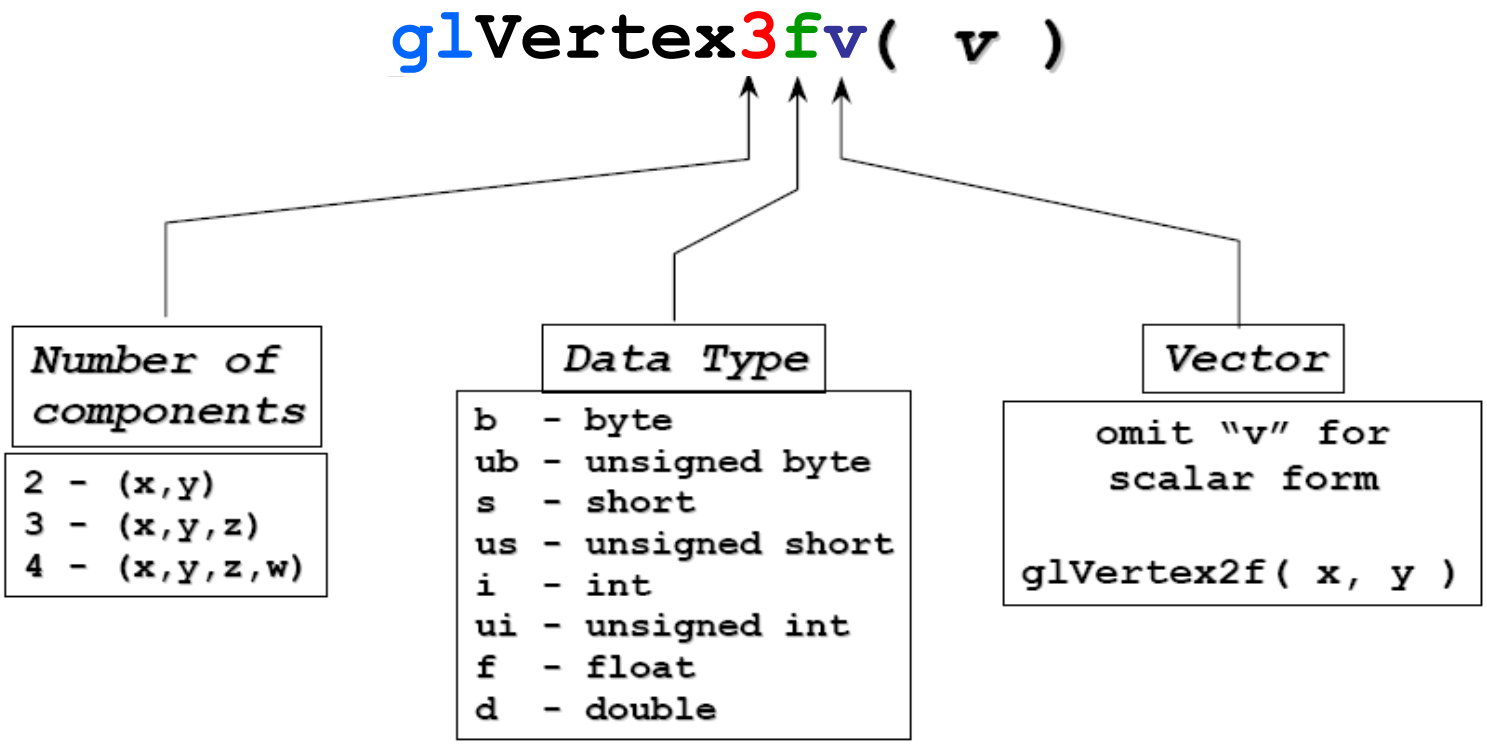


Immediate mode

Between glBegin - glEnd, the following OpenGL commands are allowed:

- glVertex*() : set vertex coordinates
- glColor*() : set current color
- glNormal*() : set normal vector coordinates (for light)
- glTexCoord*() : set texture coordinates (for texture)
- + some other less important stuff

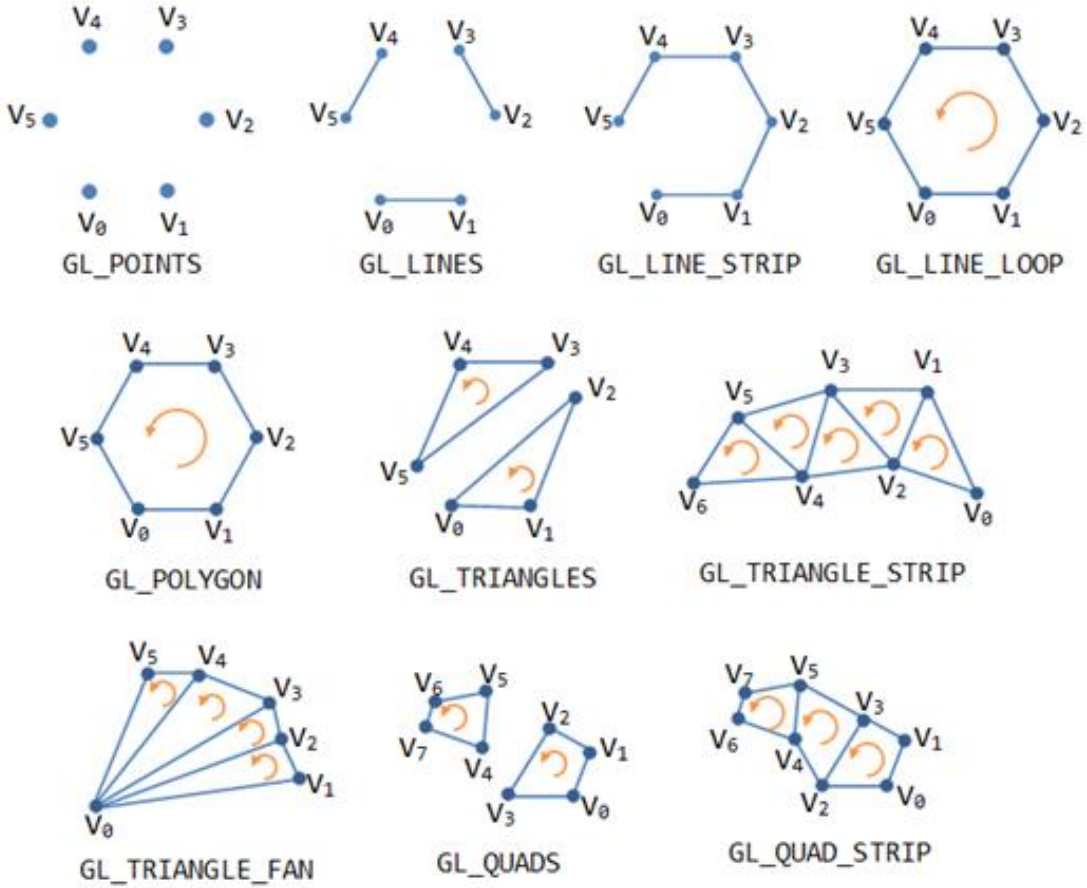
OpenGL Command Format



For `glVertex*()` calls which don't specify all the coordinates (i.e. `glVertex2f()`), OpenGL will default $z = 0.0$, and $w = 1.0$.

OpenGL Geometric Primitives

- Object geometry is specified by vertices.
- There are ten primitive types:

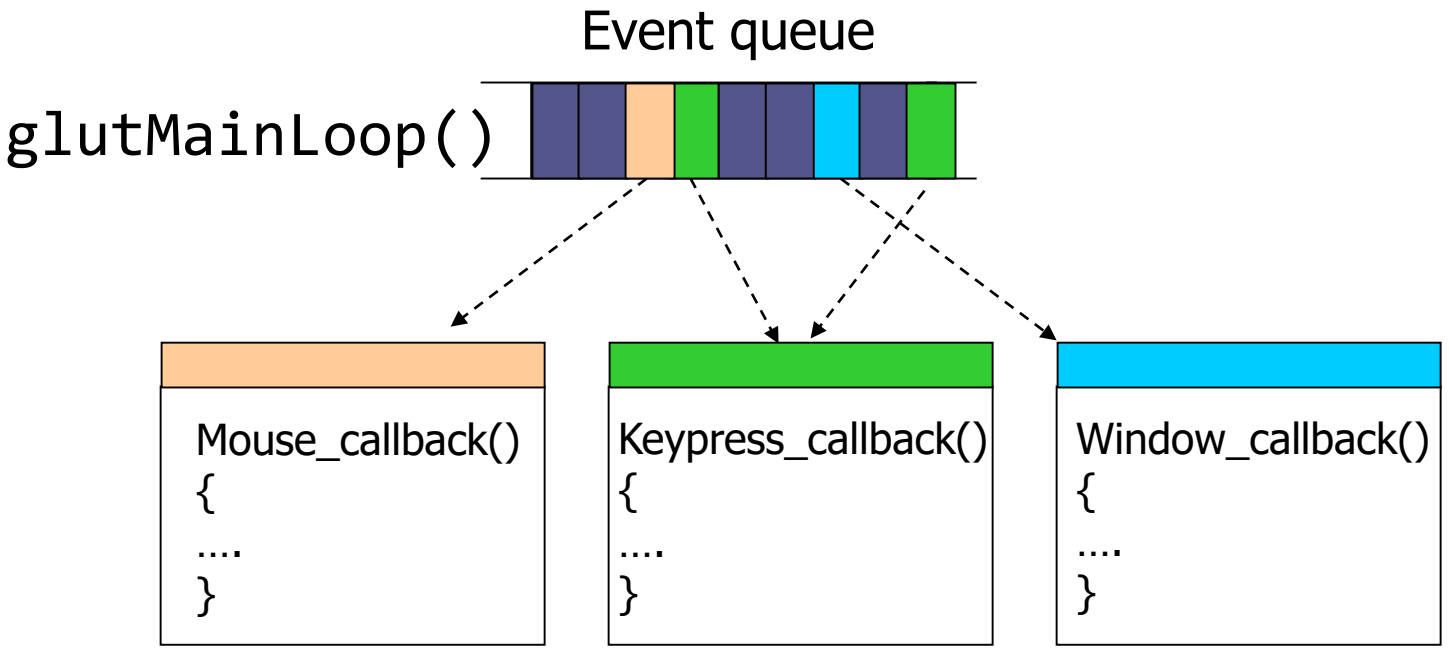


Polygon Issues

- OpenGL will only display polygons correctly that are:
  - **Simple**: edges cannot cross
  - **Convex**: All points on line segment between two points in a polygon are also in the polygon
  - **Flat**: all vertices are in the same plane
- Triangles satisfy all conditions.
  That's why they are so important for computer graphics.

Callbacks

- GLUT is event-driven
- Loop and do nothing until an event happens and then execute some pre-defined functions according to the user input



Callbacks

Event	Example	OpenGL Callback Function
Keypress	KeyDown KeyUp	`glutKeyboardFunc`
Mouse	leftButtonDown leftButtonUp	`glutMouseFunc`
Motion	With mouse press Without mouse press	`glutMotionFunc` `glutPassiveMotionFunc`
Window	Moving Resizing	`glutReshapeFunc`
System	Idle Timer	`glutIdleFunc` `glutTimerFunc`
Software	What to draw	`glutDisplayFunc`

Keyboard Callback

Captures keyboard events.

```
glutKeyboardFunc(keyfunc);
```

```
void keyfunc (unsigned char key, int x, int y)
{
   switch (key) {
     case ‘a’ : ... break;
     case ‘b’ : ... break;
   }
   glutPostRedisplay();
}
```

Mouse Callback

Captures mouse press and release events.

```
glutMouseFunc(mousefunc);
```

```
void mousefunc(int button, int state, int x, int y)
{
     if (button == GLUT_LEFT_BUTTON
        && state == GLUT_DOWN)
     {...}
     glutPostRedisplay();
}
```

Motion and PassiveMotion Callbacks

Capture mouvements of the mouse.

```
// movement without buttons pressed:
glutPassiveMotionFunc (passivemotionfunc);

// dragging:
// combine with glutMouseFunc to memorize
// which buttons are currently pressed.
glutMotionFunc(motionfunc);
```

```
void motionfunc(int x, int y)
{
   ...
    glutPostRedisplay();
}
```

Idle Callback

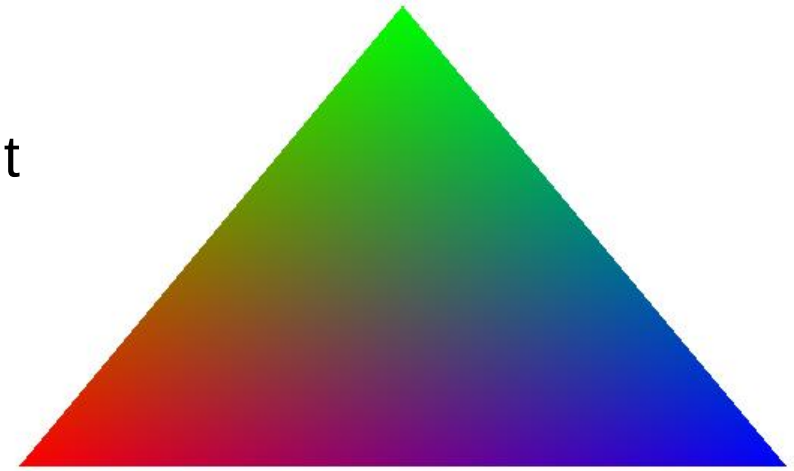
The idle callback is repeatedly called while other events are not processed. This callback is the mechanism for continuing computation and animation.

```
glutIdleFunc(idlefunc);
```

```
void idlefunc()
{
   ...
   glutPostRedisplay();
}
```

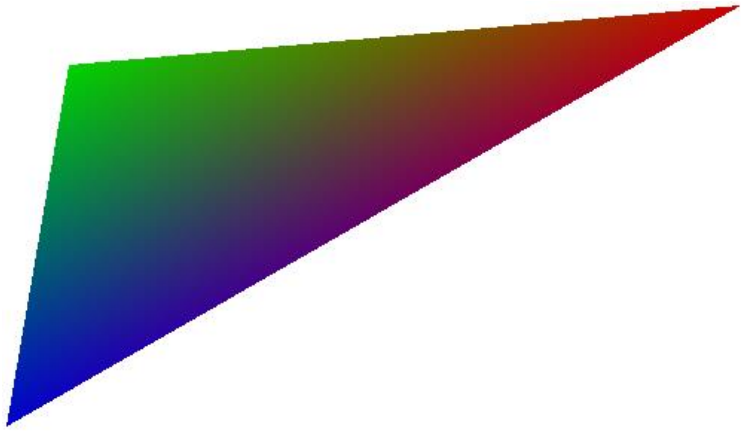
Exercise

Create a triangle with three different vertex colors. Note how OpenGL interpolates vertex data.



When the user hits ‘q’ or esc, the application quits (`glutLeaveMainLoop()`).

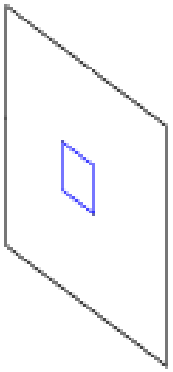
The user can drag the triangle vertex 1-2-3 with the left-middle-right button.



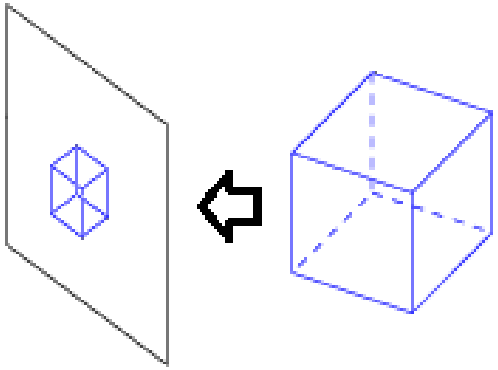
Use idlefunc to calculate the current FPS and indicate it in the window title.

3D Scenes

2D



3D



Add z coordinates to the vertices.

Homogeneous Coordinates

- Each vertex is a column vector

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

- w is 1.0
- If w is changed, we can recover x, y and z by division by w.
- All operations are matrix multiplications
- Directions can be represented with w = 0.0

Transformations

- A vertex is transformed by matrices
  - All affine operations (rotation, translation, scaling, projection) are matrix multiplications
  - For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves w unchanged.

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Transformations

- 4 steps for creating an image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Every change in coordinate systems is equivalent to a transformation matrix
  - specify geometry (model matrix)
  - specify camera (view matrix)

  } modelview matrix
  - project (projection matrix)
  - map to viewport (viewport matrix)
- All transformation matrices can be set by the programmer, but the operations are carried out within the rendering pipeline.

Working with Transformations

- Two styles of specifying transformations
  - Specify matrices
    - `glLoadMatrix`
    - `glMultMatrix`
  - Specify operation
    - `glTranslate`
    - `glScale`
    - `glRotate`
    - `glOrtho`

Modelview Transformation

The modelview matrix is used to multiply vertices at the first stage of the rendering pipeline.

```
void displayfunc()
{
      ...
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();

      // add camera position here (gluLookAt)

      glRotatef(angle, 1.0f, 0.0f, 0.0f);

      glBegin(...);
            ...
      glEnd();

      glutSwapBuffers();
}
```

Viewing Transformation

- Creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an up-vector.
- Position the camera in the scene

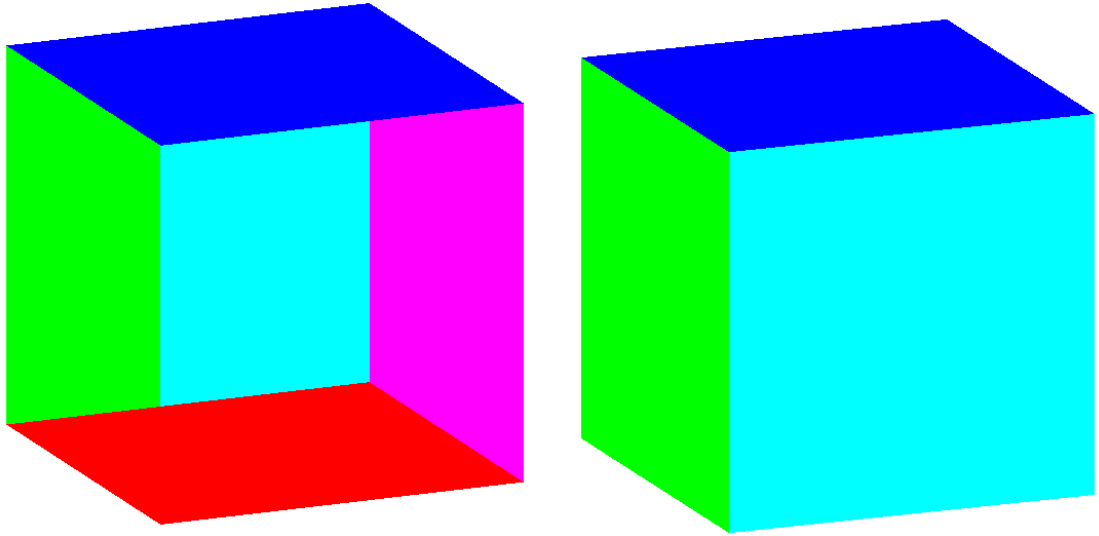
```
gluLookAt(eye_x, eye_y, eye_z,
          aim_x, aim_y, aim_z,
          up_x, up_y, up_z)
```

- `gluLookAt()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)` and `glLoadIdentity()`.

- Because of degenerate positions, gluLookAt() is not recommended for animated fly-over applications. An alternative is to specify a sequence of rotations and translations that are concatenated with an initial identity matrix.

Exercise

- Create a new scene
- Draw a cube with 6 colored sides
- Mouse drag rotates the cube
- Whoops, we need a depth test!



Depth Buffering

Request a depth buffer

```
glutInitDisplayMode
  (GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH);
```

Enable depth buffering

```
glEnable(GL_DEPTH_TEST);
```

Clear color and depth buffers

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

Render scene

Swap color buffers

Display Lists

Immediate mode is bad. You are retransmitting OpenGL commands over and over again.

A display list is a group of OpenGL commands stored and compiled on the GPU for later execution and reuse.

```
// global variable:
GLuint DrawListCube;
...

// in your init method:
DrawListCube = glGenLists(1);
glNewList(DrawListCube, GL_COMPILE);
drawCube(); // contains glBegin ... glEnd
glEndList(); // the display list is now loaded in the GPU
...
// in your displayfunc:
glCallList(DrawListCube);
...
// in your cleanup:
glDeleteLists(DrawListCube, 1);
```

Exercise

Change immediate mode to display list.

The matrix stack

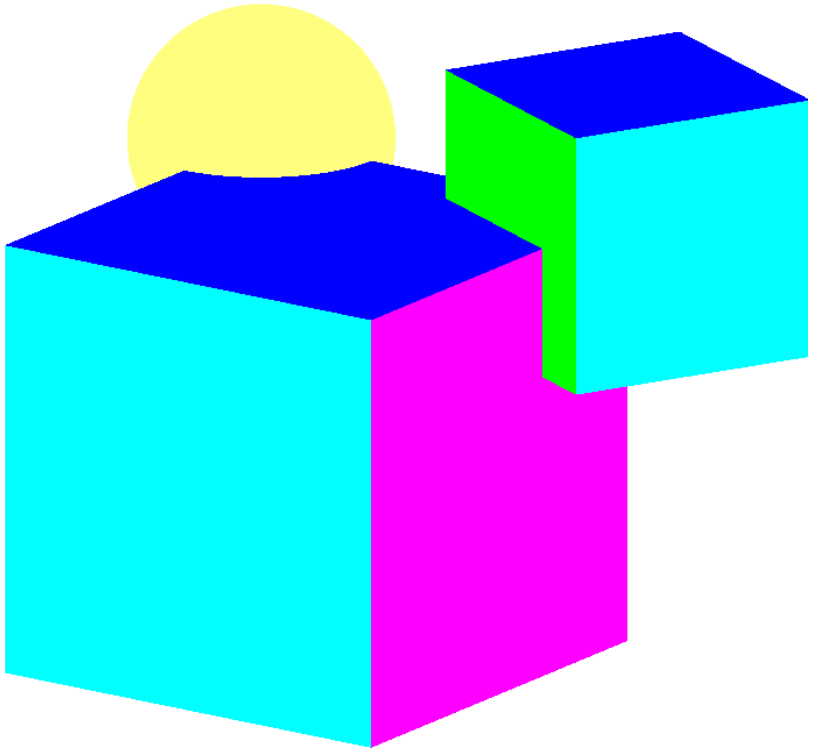
- `glPushMatrix` pushes the current matrix stack down by one, duplicating the current matrix. That is, after a `glPushMatrix` call, the matrix on top of the stack is identical to the one below it.
- `glPopMatrix` pops the current matrix stack, replacing the current matrix with the one below it on the stack
- Ideal for hierarchical models

Exercise

Create a 2nd cube and a sphere.

Attach them to the first cube.

Animate the 2nd cube.

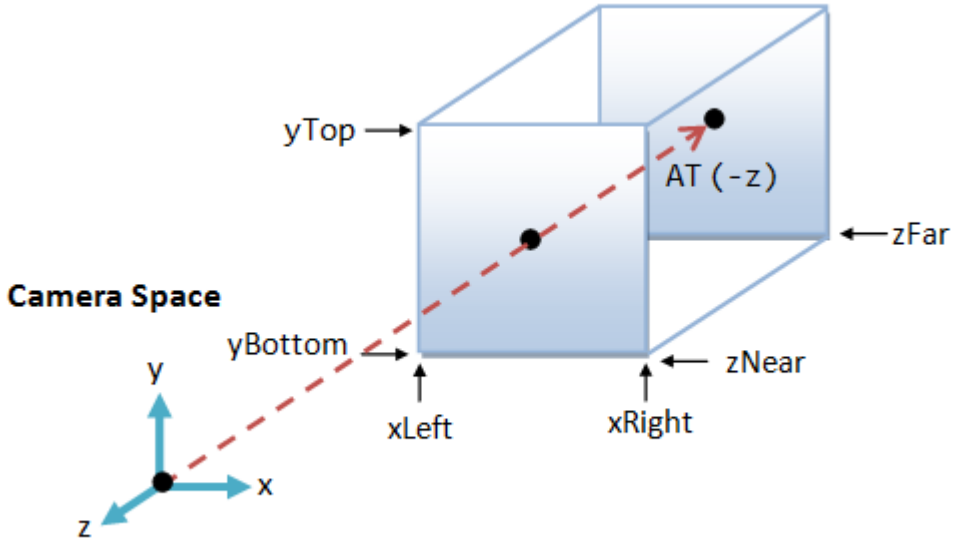


Projection Transformation

Orthographic parallel projection

`glOrtho(left, right, bottom, top, zNear, zFar)`

The view volume is a parallelepiped.



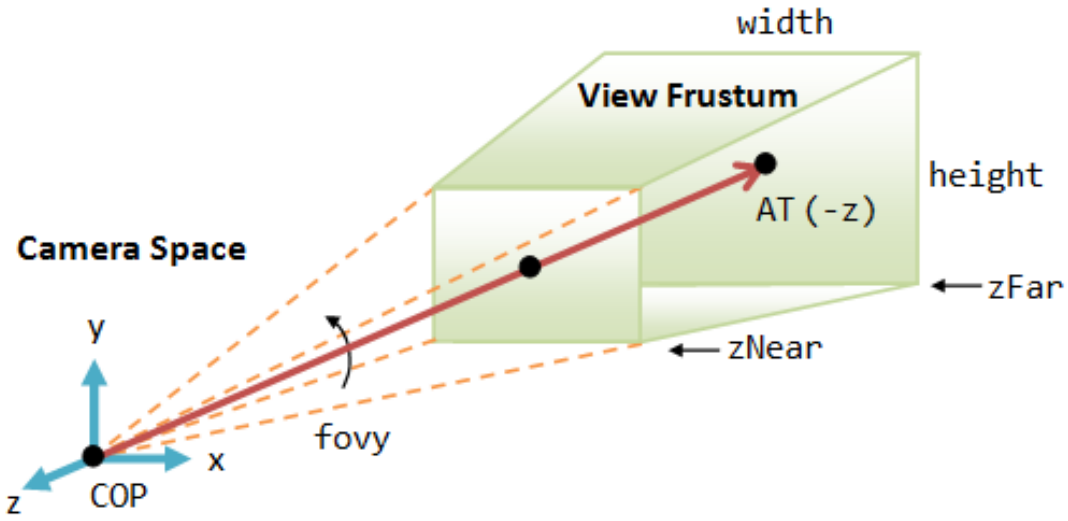
Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Projection Transformation

Perspective projection

`gluPerspective(fovy, aspect, zNear, zFar)`

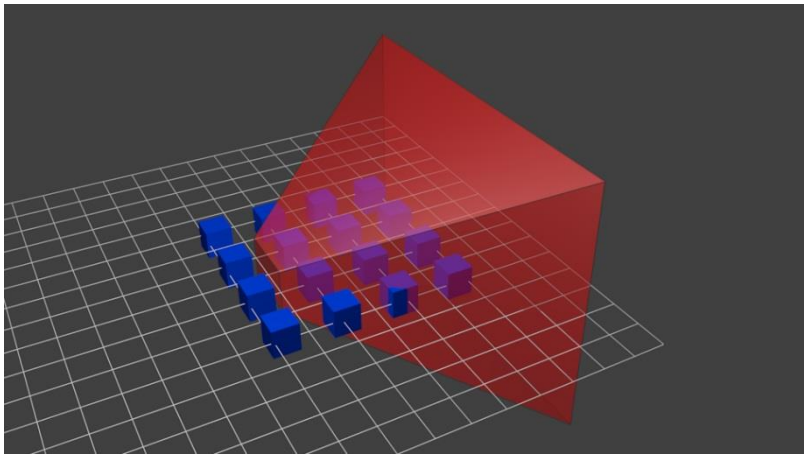
The view volume is a frustum.



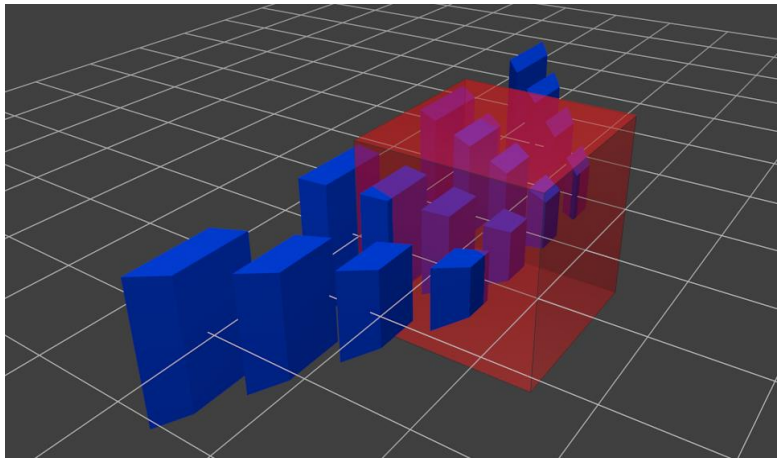
Projection Transformation

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(…) or gluPerspective(…);
```

The projection matrix converts the view volume into Normalized Device Space: all vertices defined in a small cube. Everything inside the cube is onscreen.



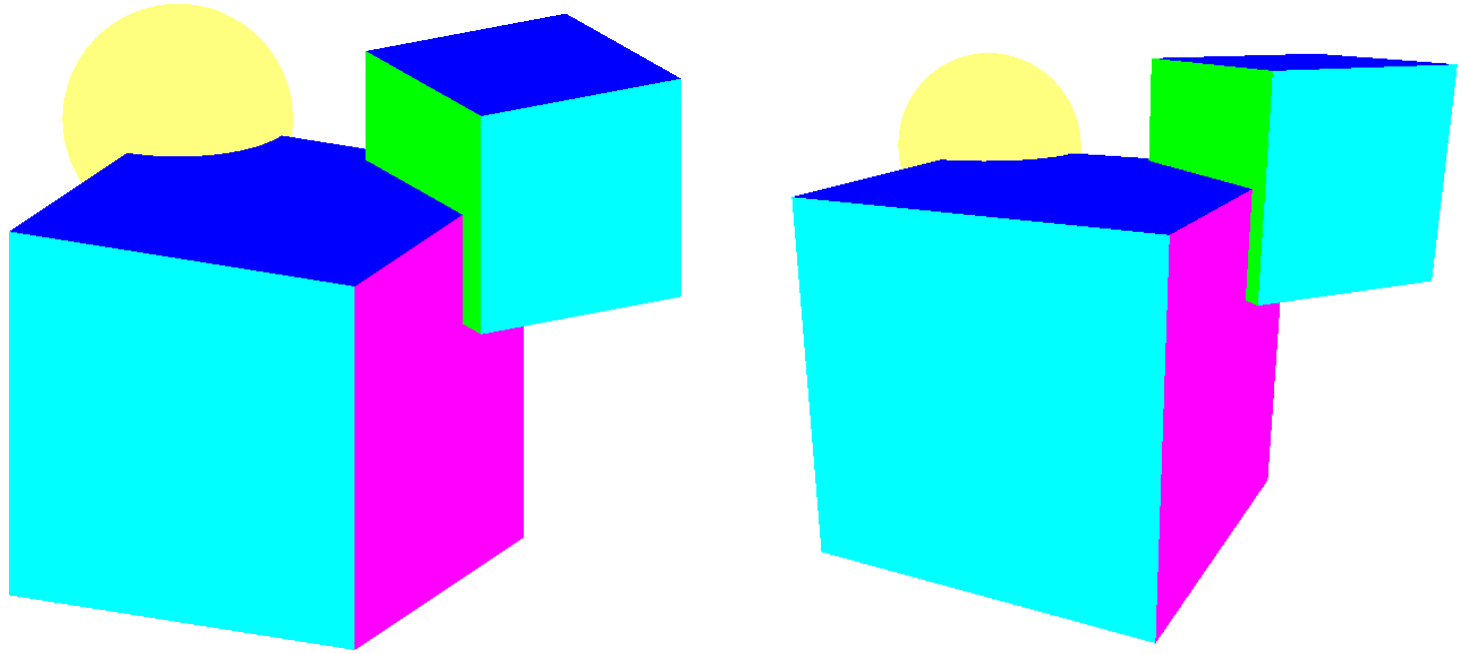
View volume



Normalized device space

Exercise

Toggle between orthographic and perspective projection.

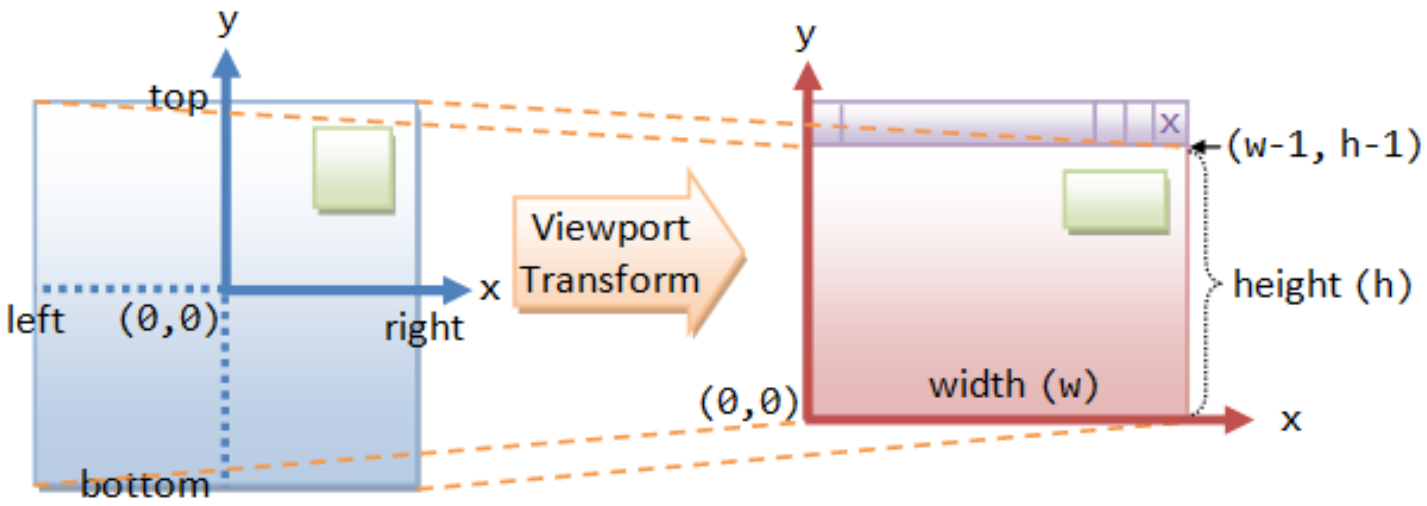


Viewport Transformation

The viewport matrix converts the Normalized Device Space into screen pixels.

```
glViewport(0, 0, SCREEN_X, SCREEN_Y);
```

specifies the area of the window that the drawing region should be put into. Note that if the viewport does not work with the same aspect (w/h) as the projection, the image is distorted.



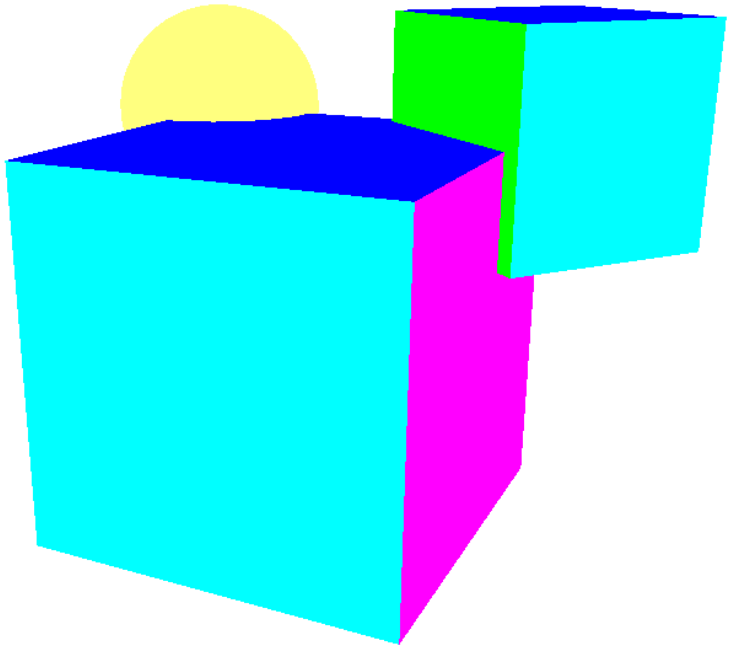
Exercise

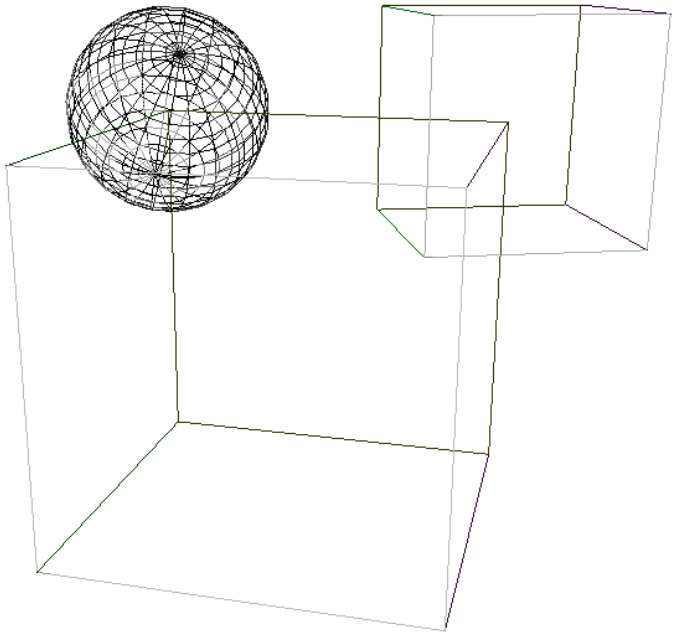
Adjust the projection so that the view is not distorted when the window is resized (`glutReshapeFunc`)

- Update Viewport
- Update projection matrix

Add-on I: Wireframe

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

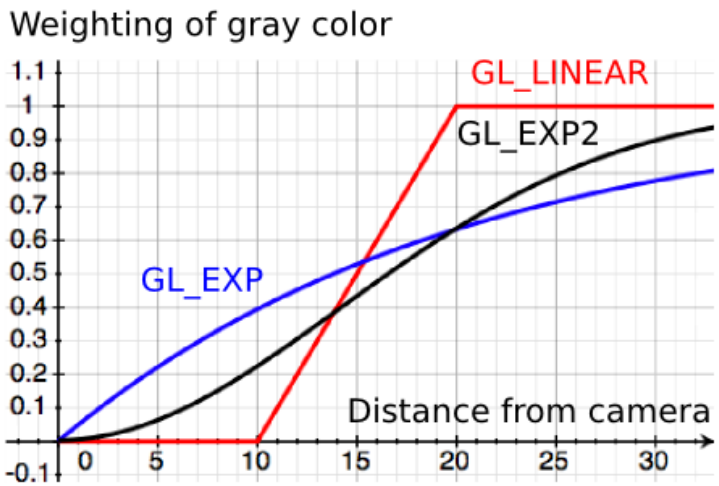


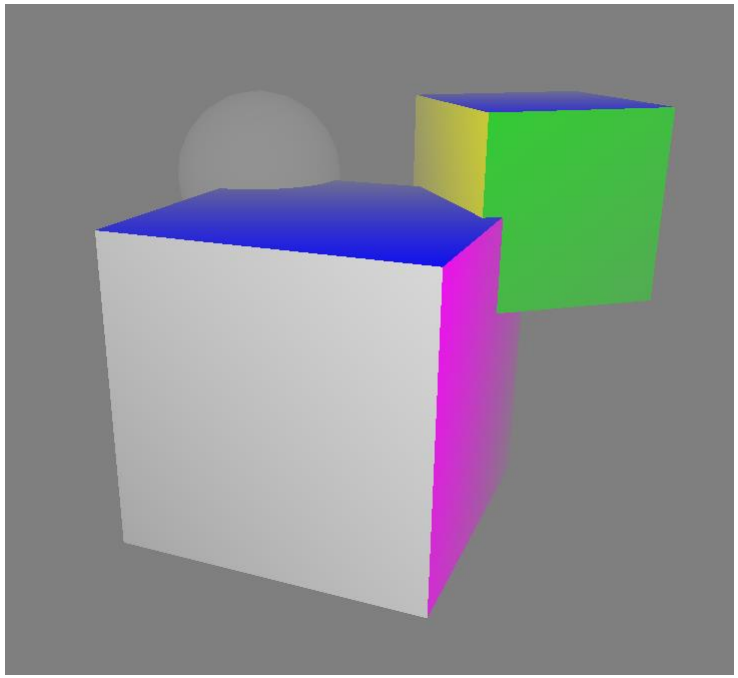


Exercise: Add wireframe mode to the scene

Add-on II: Fog

```
glEnable(GL_FOG);
GLfloat fogColor[] = {0.5f, 0.5f, 0.5f, 1};
glFogfv(GL_FOG_COLOR, fogColor);
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogf(GL_FOG_START, 10.0f);
glFogf(GL_FOG_END, 20.0f);
```





Exercise: Add fog to the scene

In fog mode, don't forget to change the background color to the fog color!

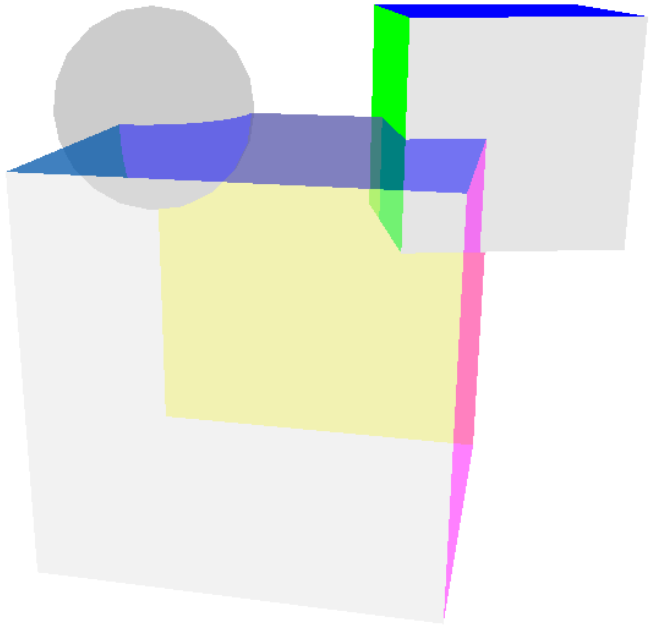
Add-on III: Alpha blending

Blending can be used to make objects appear transparent.

Caution! The Z-buffer does not work correctly for transparent polygons. Draw opaque objects first!

Errors might still arise when you render one translucent polygon behind another. This is a difficult sorting problem.

```
glColor4f(1.0, 1.0, 0.0, 0.5f);
// 4th parameter is alpha
...
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
   GL_ONE_MINUS_SRC_ALPHA);
```



Exercise: Add translucency to the maincube

Your stunning OpenGL1 demo

Create a demo application with all the features we have seen (exercises and add-ons).