OpenGL
I –GLUT & Fixed Function Pipeline

Stefan BORNHOFEN
EISTI

What is OpenGL?

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

Competitors

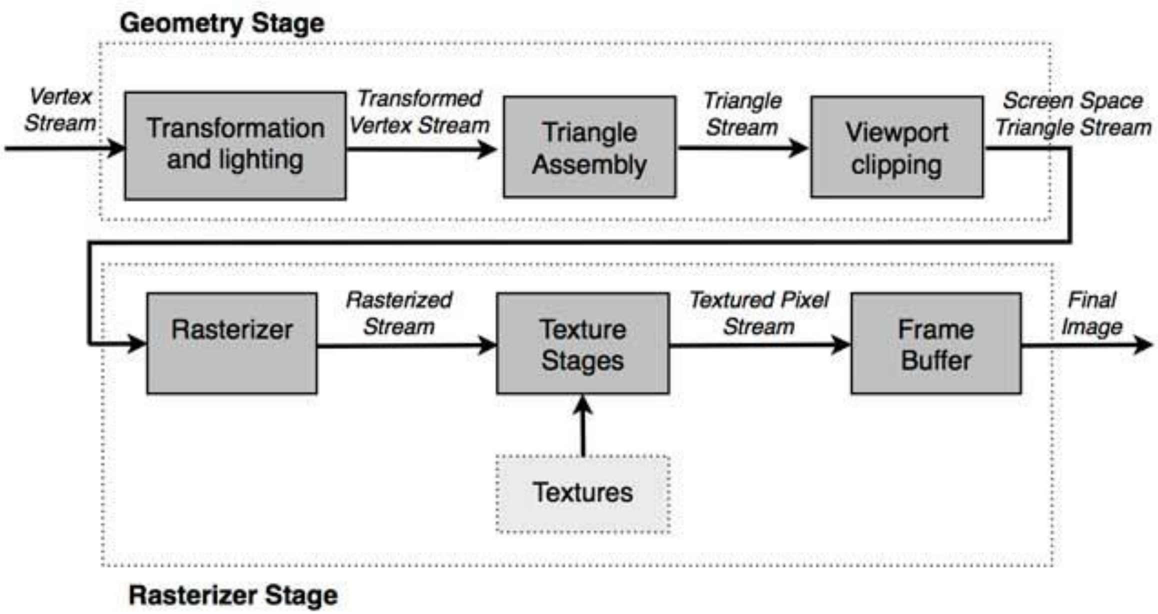
- Direct3D: a proprietary API by Microsoft that provides equivalent functionalities for use on the Windows platform
- Vulkan: “Next generation OpenGL" released in 2016 by the Khronos group. Intended to provide a variety of advantages over OpenGL (unifying OpenGL and OpenGL ES, lower overhead, better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

Related Libraries

- **GLU (OpenGL Utility Library)**
  - Offers some high level operations
  - NURBS, tessellators
  - More primitives (quadrics)
  - A simpler viewing mechanism
- **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

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.

Let's go! Install GLUT

- Download freeglut 3.0.0 for MSVC http://www.transmissionzero.co.uk/software/freeglut-devel/
- Copy "include" to "VisualStudio2013\VC\include"
- Copy "lib" to "VisualStudio2013\VC\lib"
- Copy "bin" to "C:\Windows\System32"
- Create a Visual Studio 2013 Project
  - Win32 console application
  - Add "opengl32.lib;freeglut.lib;" to additional dependencies

GLUT Basics

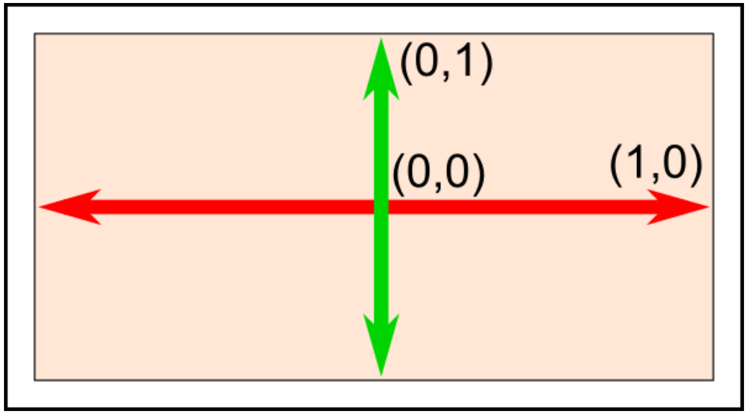
- Initialize GLUT and open window
- Initialize OpenGL state
- Register callback functions
  - render
  - keyboard
  - mouse
  - etc.
- Enter event processing loop

```
#include "stdafx.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);
	init();
	glutDisplayFunc(displayfunc);
	glutMainLoop();
}
```

```
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);
}
```

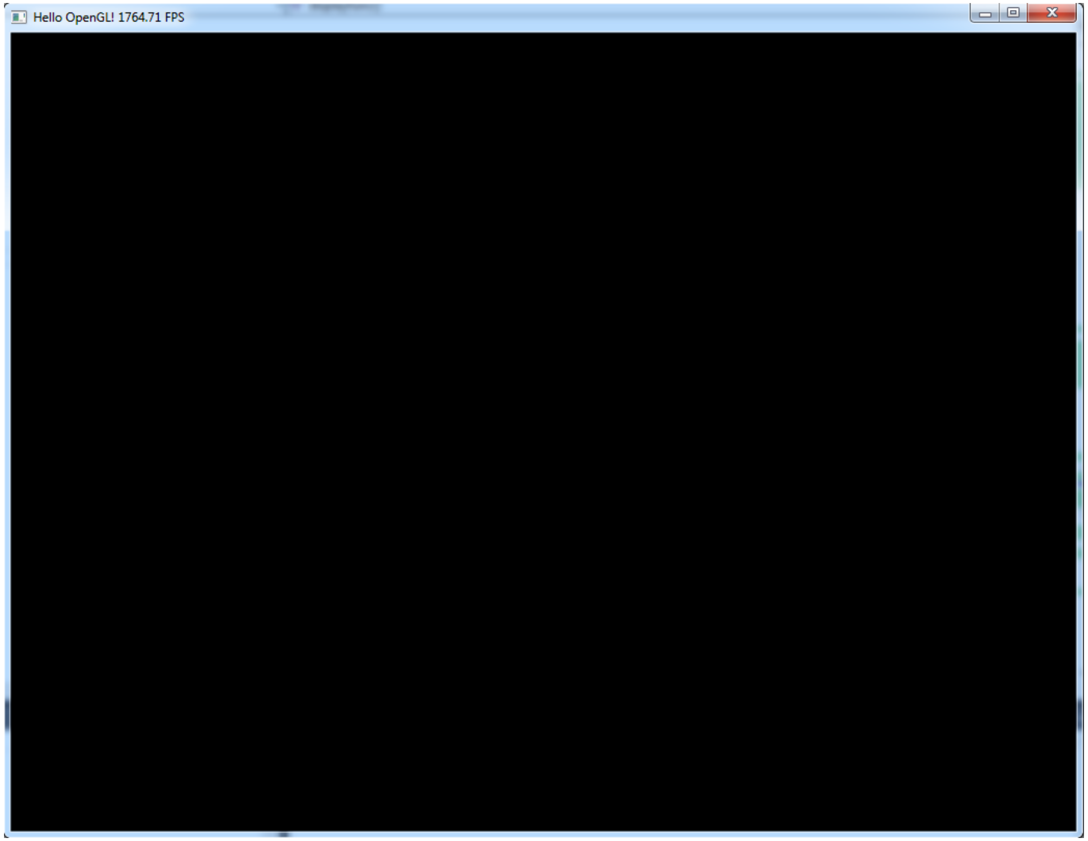


The OpenGL coordinate system is different from the window system

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

Exercise

Create a black window



Rendering Callback

- 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();
}
```

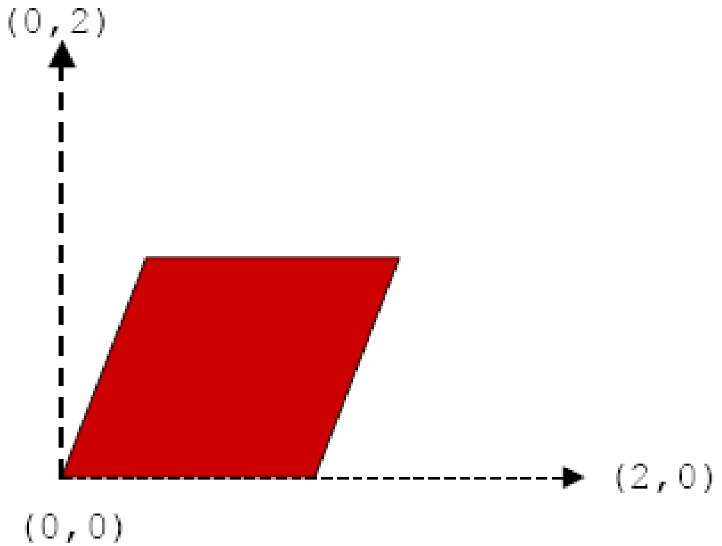
Immediate mode

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();
}
```

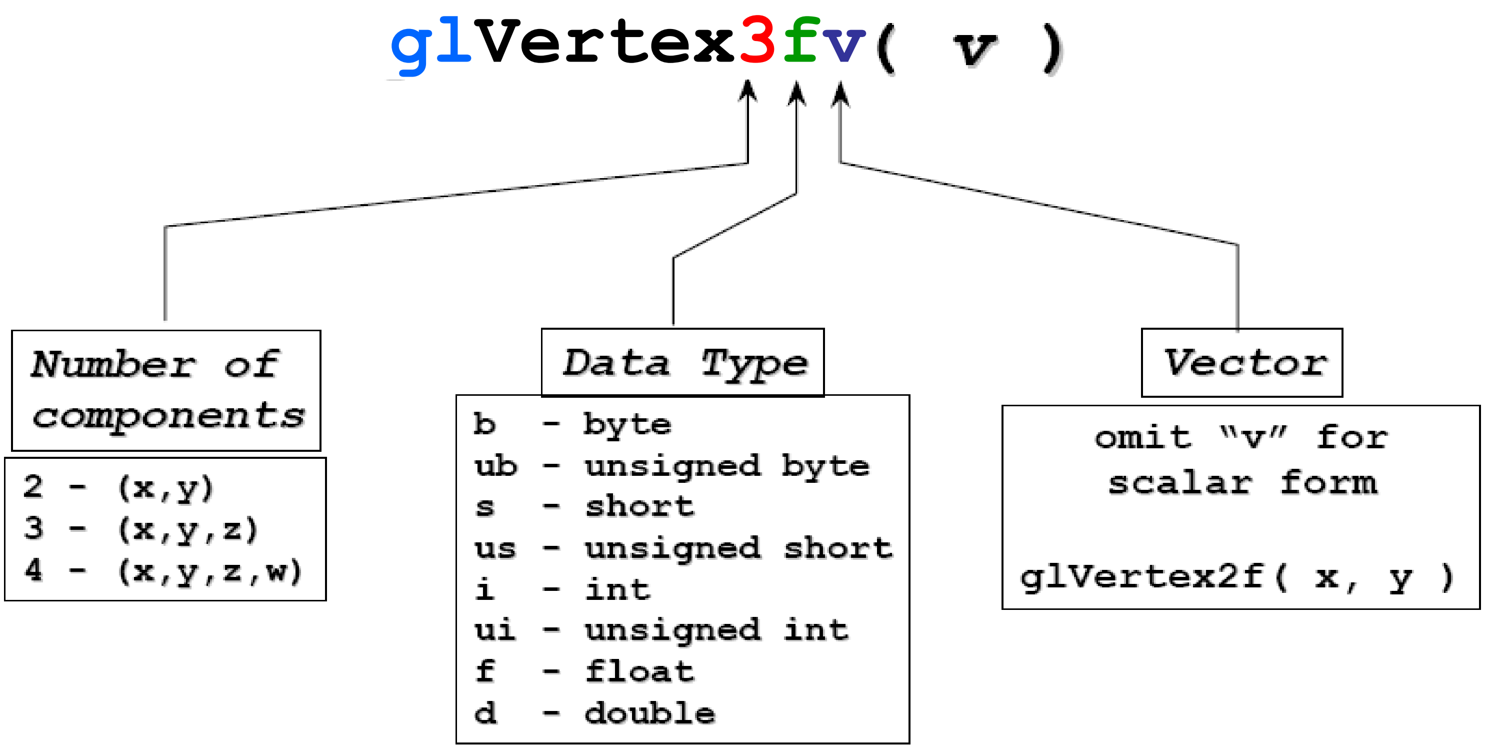


Immediate mode

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

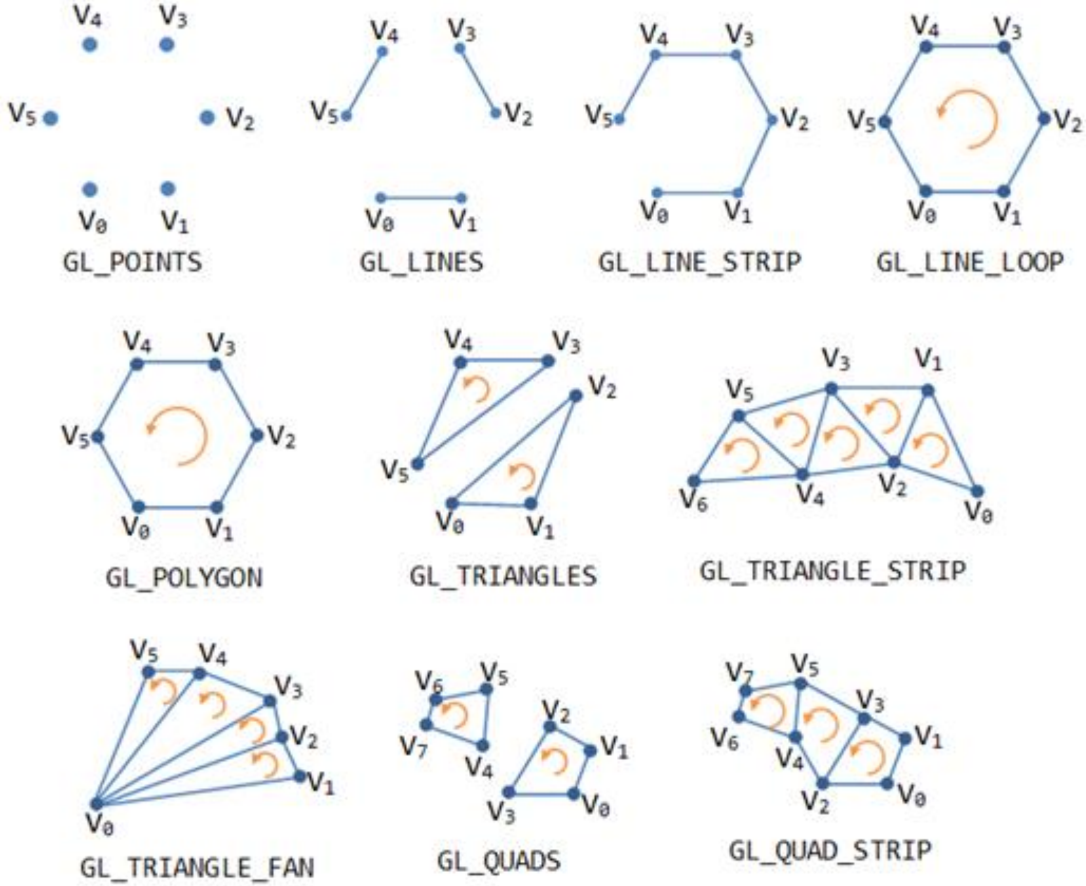
OpenGL Command Format



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:



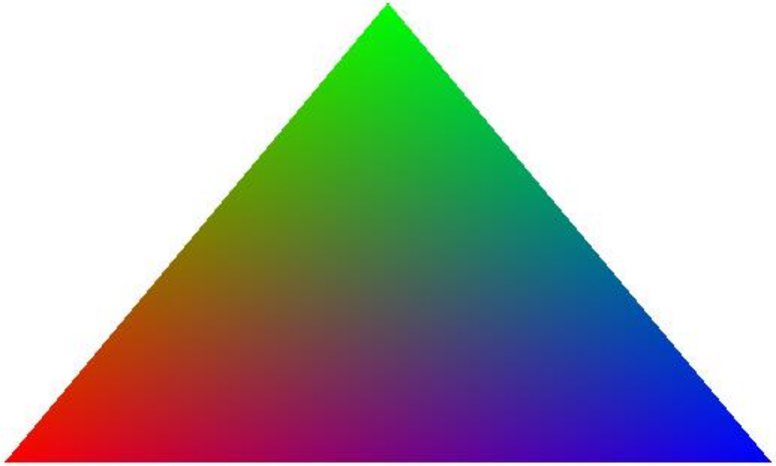
Polygon Issues

- 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.

Exercise

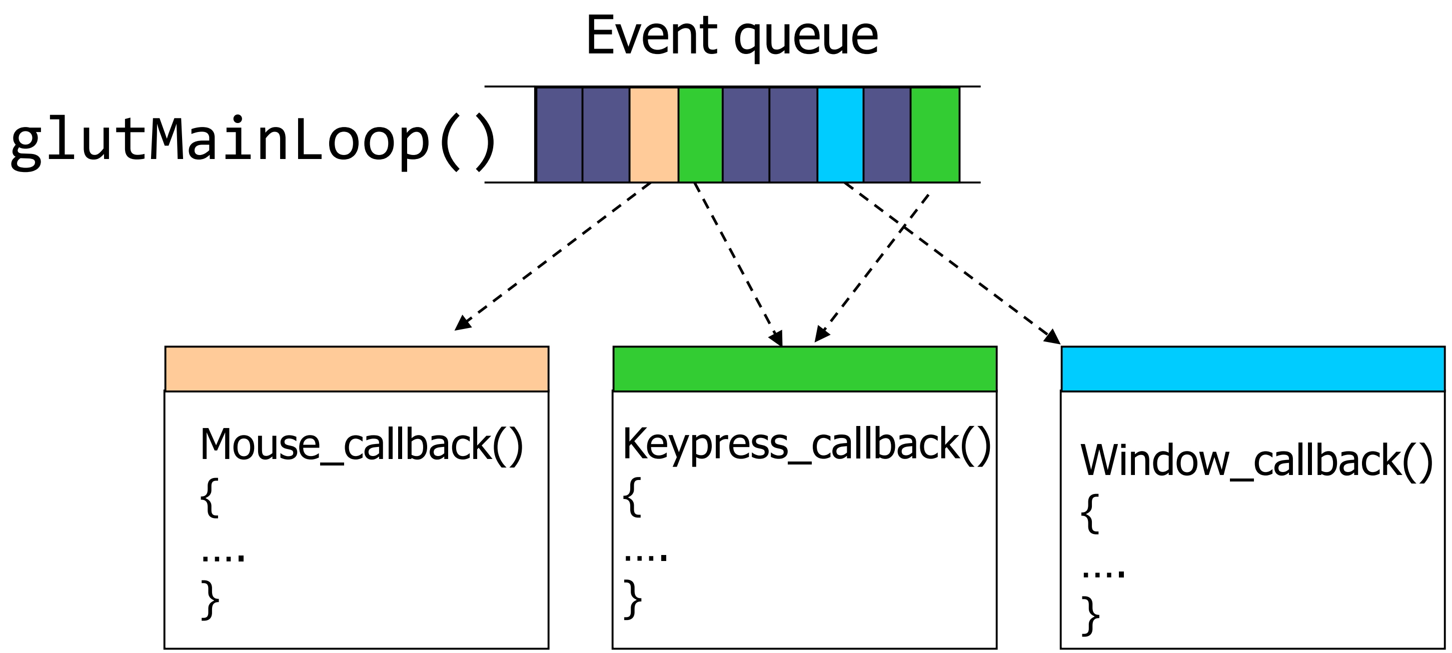
Create a colored triangle.

Note how OpenGL interpolates vertex data within the polygon.



More callbacks!

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



Register callbacks for each event

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

Keyboard Callback

```
glutKeyboardFunc(keyfunc);

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

Exercice

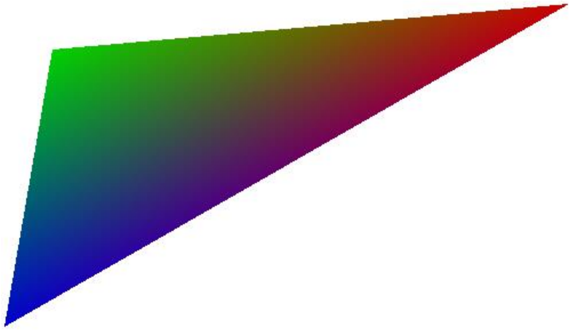
When the user hits 'q' or esc, the application quits.

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)
    {...}
}
```



Exercise

When the mouse is left-middle-right clicked, triangle vertex 1-2-3 jumps to the mouse position

Motion Callback

Captures mouse drag.

```
glutMotionFunc(motionfunc);

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

Exercise
Triangle vertex 1-2-3 follows mouse drag.

Idle Callback

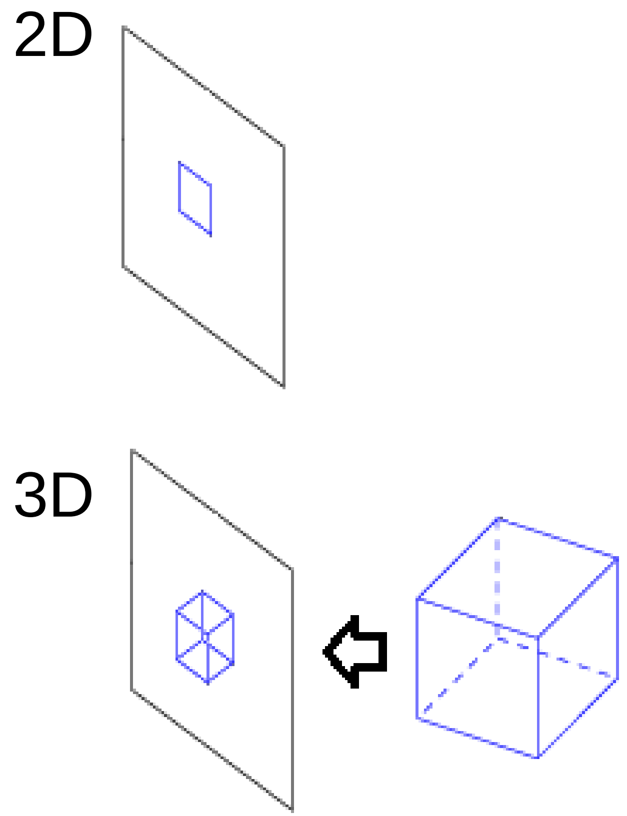
For continuous update.

```
glutIdleFunc(idlefunc);

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

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

3D Scenes



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.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(angle, 1.0f, 0.0f, 0.0f);
glBegin(...);
     ...
glEnd();
```

Viewing Transformation

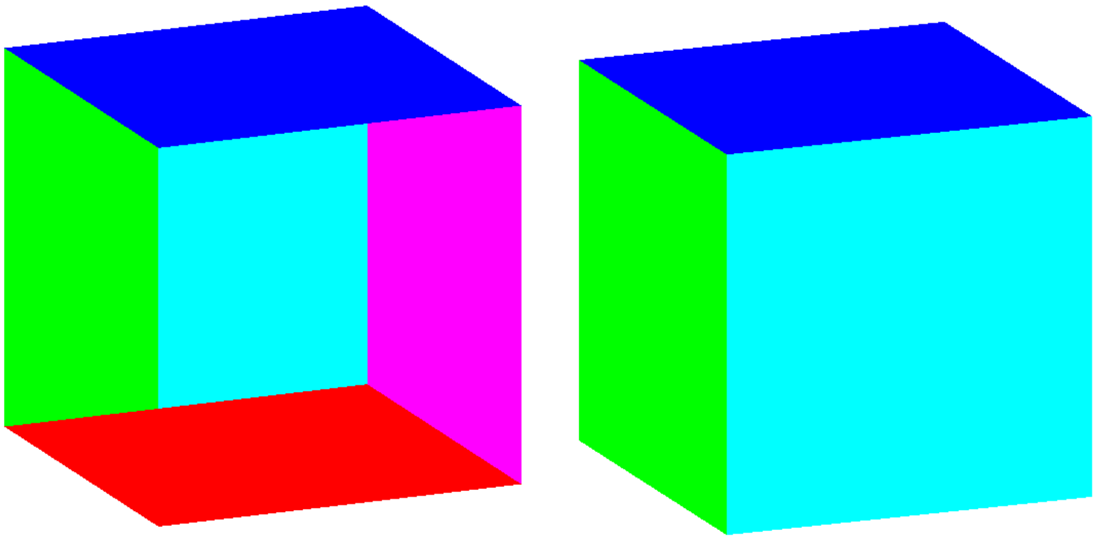
- Creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an up-vector.
- Position the camera in the scene

$$\texttt{gluLookAt}(eye_x,\ eye_y,\ eye_z,\ aim_x,\ aim_y,\ aim_z,\ up_x,\ up_y,\ up_z)$$

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

Exercise

- Create a new project
- Draw a cube with 6 colored sides (type 1 and 2 to toggle between triangle scene and the cube scene)
- 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.

```
GLuint DrawListCube = glGenLists(1);

glNewList(DrawListCube, GL_COMPILE);
drawCube();
glEndList();
// display list is now loaded in the GPU
...
glCallList(DrawListCube);
...
// when you are done:
glDeleteLists(DrawListCube, 1);
```

Exercise

Change immediate mode to display list.

The matrix stack

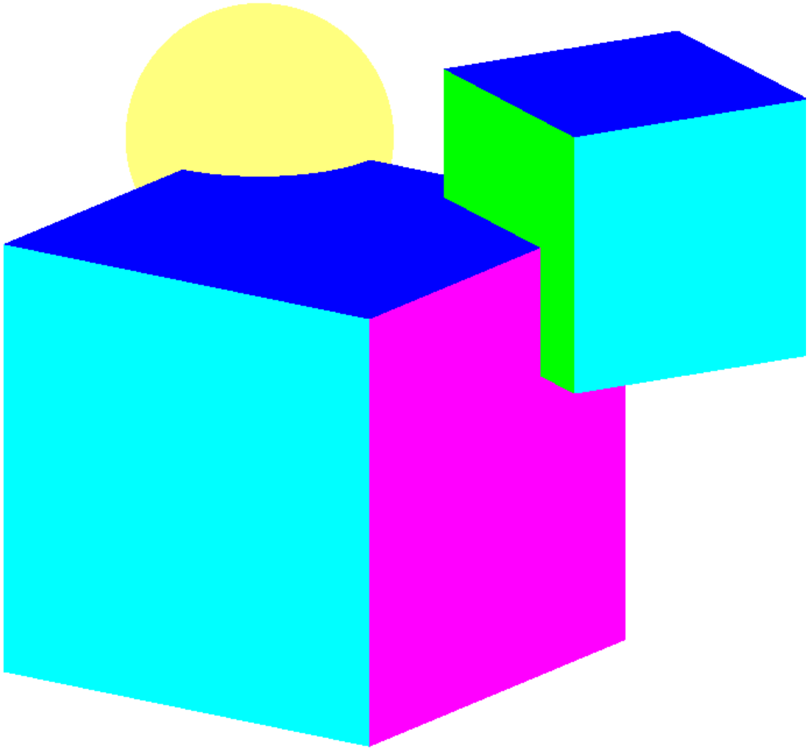
- `glPushMatrix` pushes the current matrix stack down by one, duplicating the current matrix. That is, after a `glPushMatrix` call, the matrix on top of the stack is identical to the one below it.
- `glPopMatrix` pops the current matrix stack, replacing the current matrix with the one below it on the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

Exercise

Create a 2nd cube and a sphere.

Attach them to the first cube.

Animate the 2nd cube.

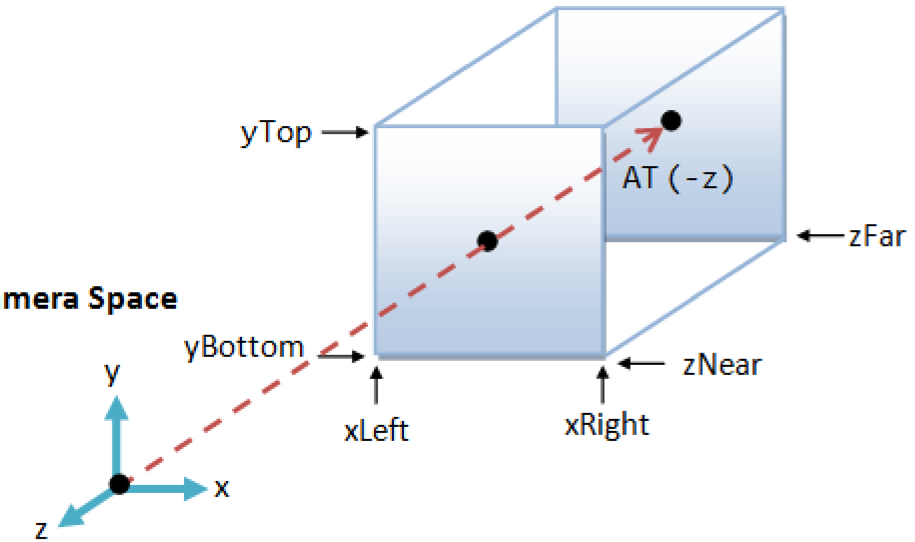


Projection Transformation

Orthographic parallel projection

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

The view volume is a parallelepiped.



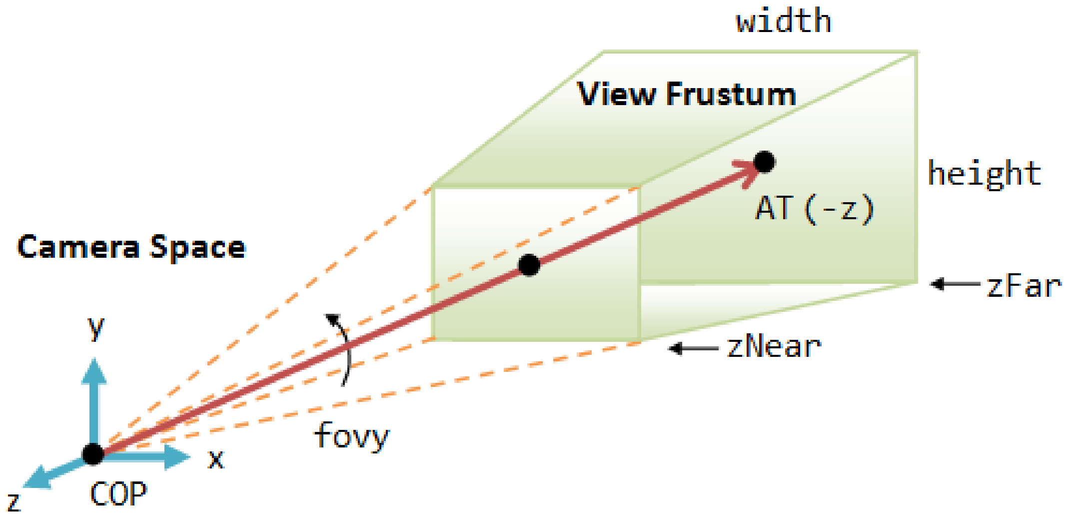
Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Projection Transformation

Perspective projection

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

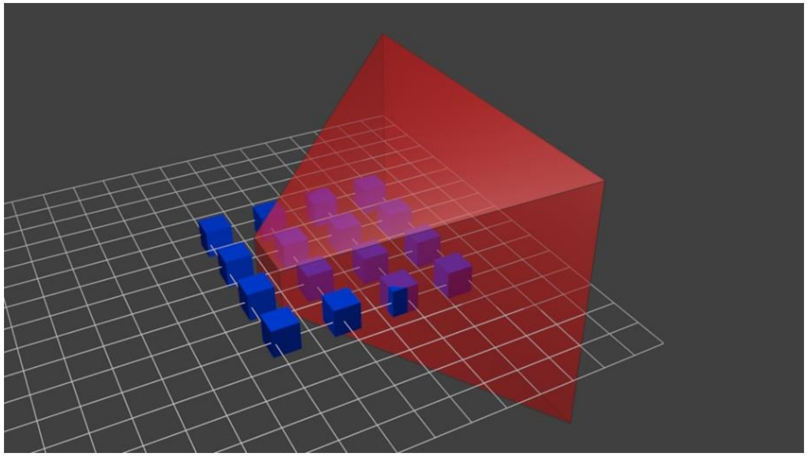
The view volume is a frustum.



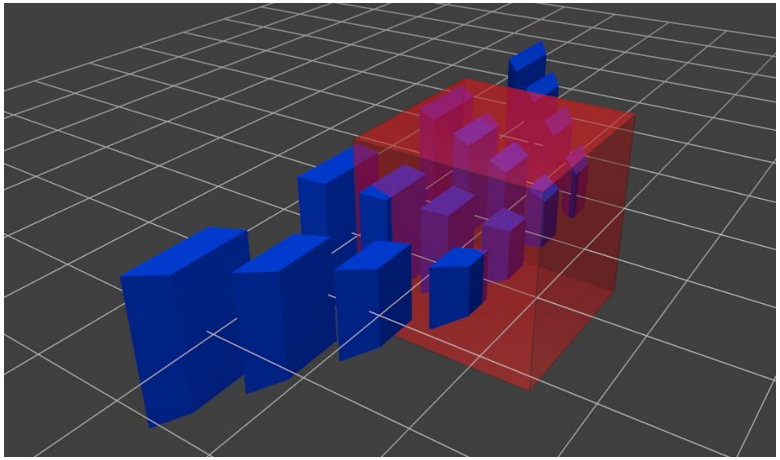
Projection Transformation

```
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.



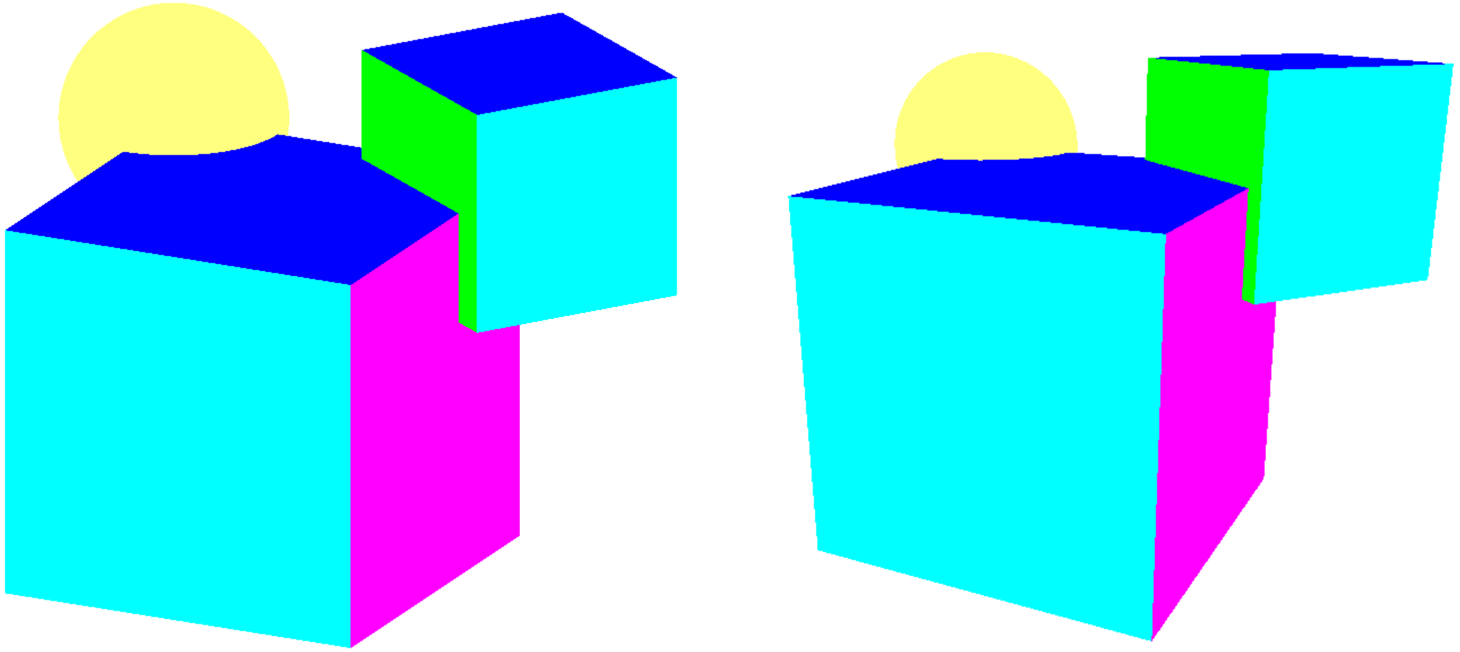
View volume



Normalized device space

Exercise

Toggle between orthographic and perspective projection.

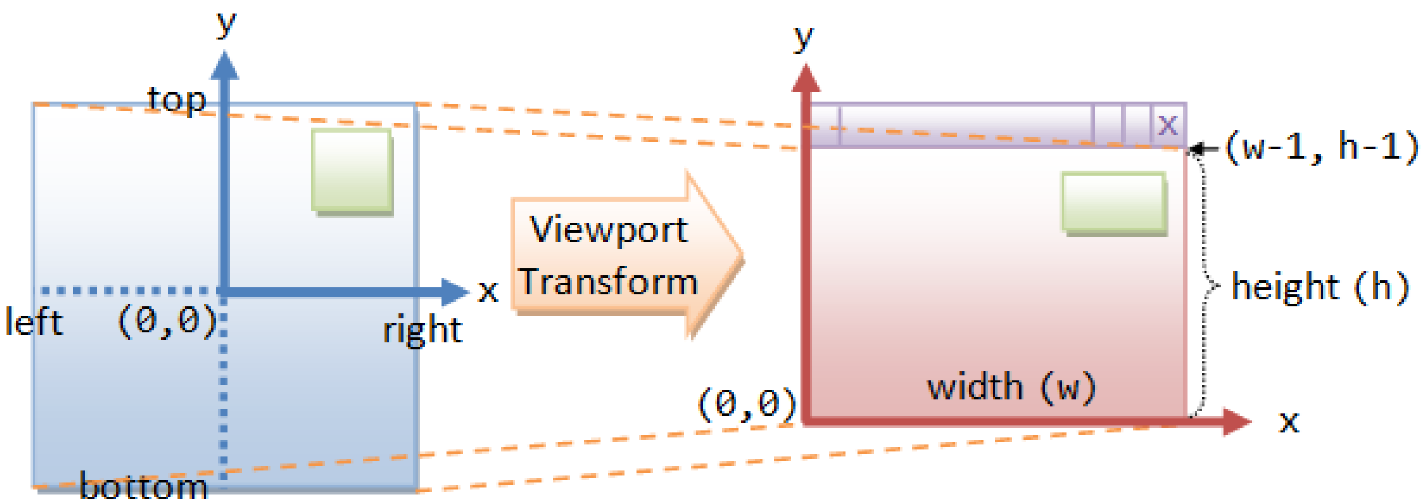


Viewport Transformation

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.

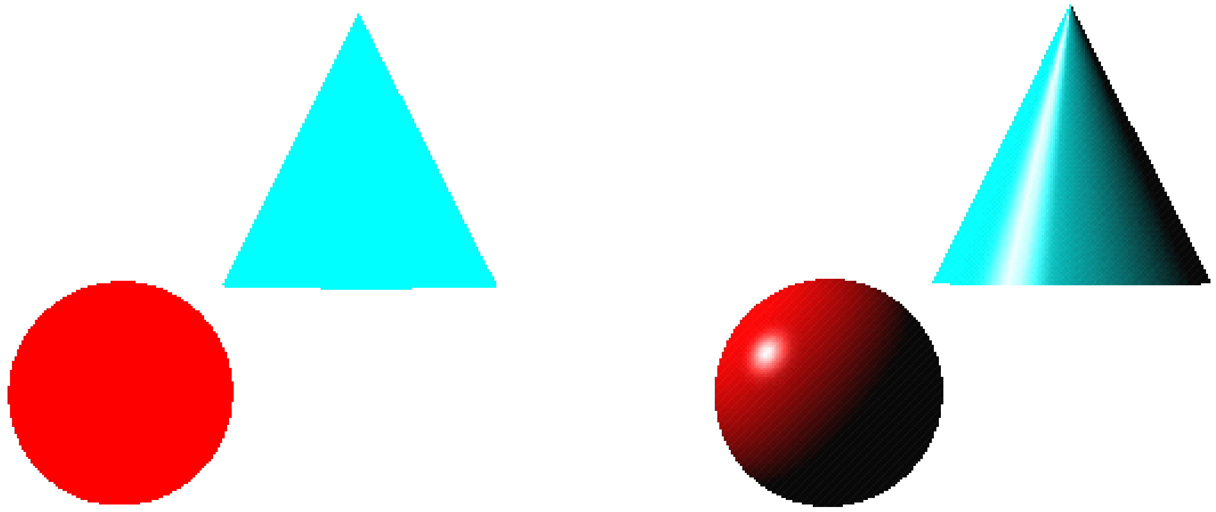


Exercise

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

Lighting

- Lighting simulates how objects reflect light based on several properties
  - surface normals and material
  - light properties: color and position
  - global lighting parameters: ambient light



Surface Normals

- Normals define how a surface reflects light

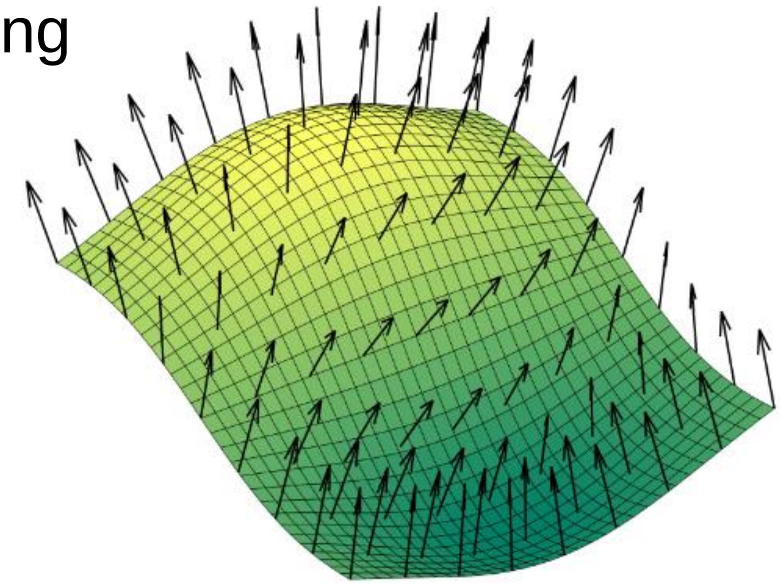
```
glNormal3f(x, y, z)
```

for all vertices until a new normal is provided

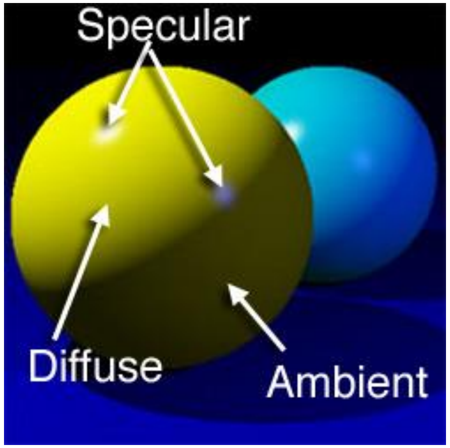
- Use unit normals for proper lighting.

Warning: Scaling (`glScale`) also affects the normal lengths. If you scale and lighting is on, enable automatic normalization of normals by calling

```
glEnable(GL_NORMALIZE)
```

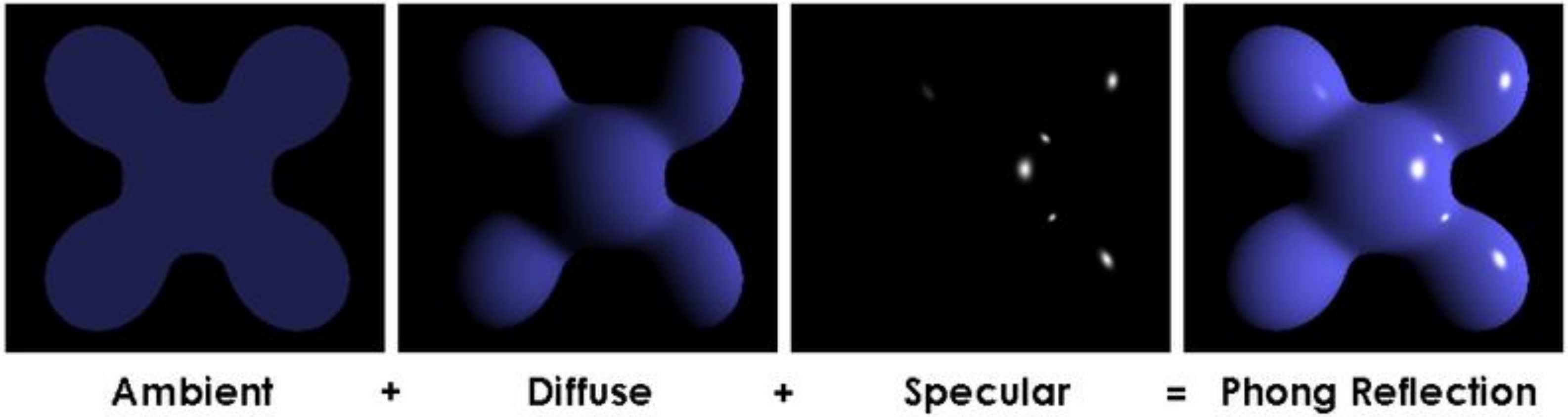


Phong lighting model



OpenGL divides lighting into three parts

- Ambient is a constant.
- Diffuse depends on the angle between the light vector and the surface normal vector.
- Specular depends on the angle between the eye vector and the surface normal vector.



Material Properties

Define the surface properties of a primitive

- `glMaterialfv(face, property, value);`

GL_DIFFUSE	Base color
GL_SPECULAR	Highlight Color
GL_AMBIENT	Low-light Color
GL_EMISSION	Glow Color
GL_SHININESS	Surface Smoothness

- `face: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.`
  It is possible to set different materials for front and back.

Light Properties

`glLightfv (light, property, value);`

- `light` specifies the light index
  - multiple lights possible, starting with `GL_LIGHT0`
  - `0...GL_MAX_LIGHTS – 1`
- `properties`
  - colors
  - position and type
  - attenuation

Examples

```
GLfloat light_pos[] = { 0.0, 0.0, 1.0, 0.0 };
GLfloat ambient_light[] = { 0.01f, 0.01f, 0.01f, 1.0f };
GLfloat diffuse_light[] = { 0.6f, 0.6f, 0.6f, 1.0f };
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient_light);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_light);
```

Types of Light

- OpenGL supports two types of Lights
  - Infinite (Directional) light sources
  - Local (Point) light sources

$$w = 0 \quad \text{Infinite Light directed along} \begin{pmatrix} x & y & z \end{pmatrix}$$

$$w \neq 0 \quad \text{Local Light positioned at} \begin{pmatrix} x/w & y/w & z/w \end{pmatrix}$$

- The type of light is controlled by w coordinate
- A light position is transformed by the current ModelView matrix when it is specified

Controlling the Light Position

- Different effects based on the current modelview matrix.
  1) Modelview = identity matrix: The light remains fixed relative to the eye, like a headlight.
  2) *ModelView = viewing matrix only:* the light appears to be fixed in the scene, like a lamppost.
  3) *Modelview = object modelview matrix:* the light position remains relative to a given object
  4) Modelview = anything: allows for arbitrary, and even animated, light positions (use push and pop).

Turning on the Lights

- Flip the switch

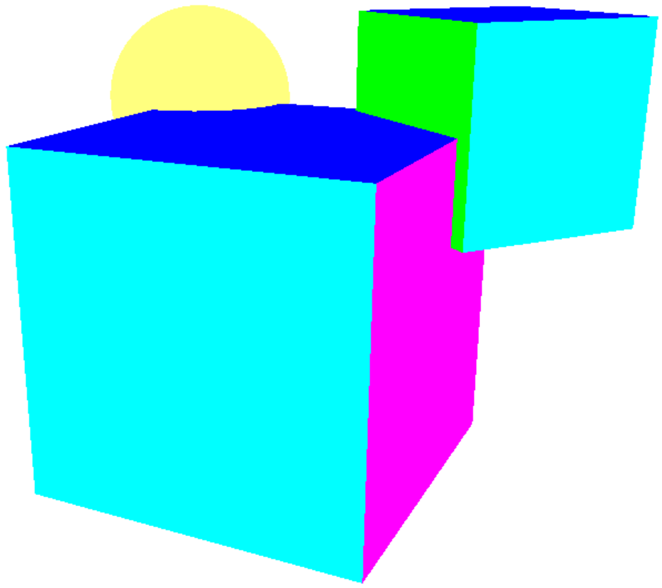
```
glEnable(GL_LIGHT<N>);
```

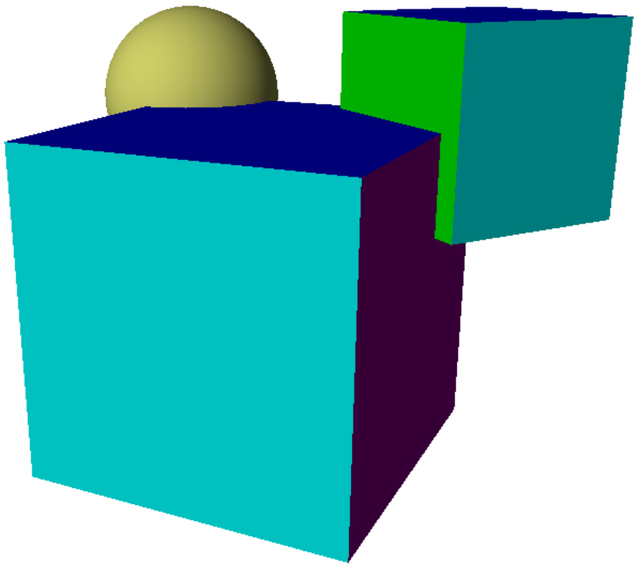
- Turn on the power

```
glEnable(GL_LIGHTING);
```

Exercise

- Add light to the scene
- The mouse controls the position of the light source

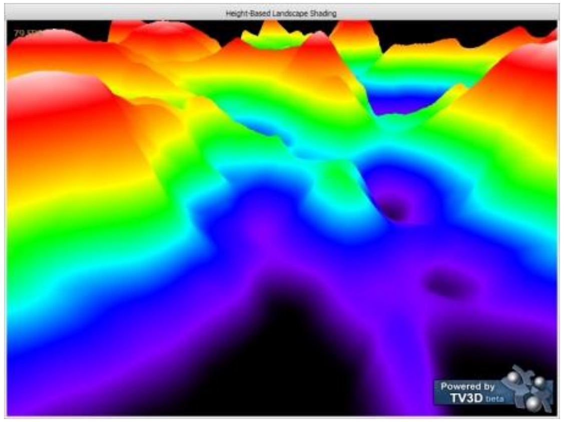


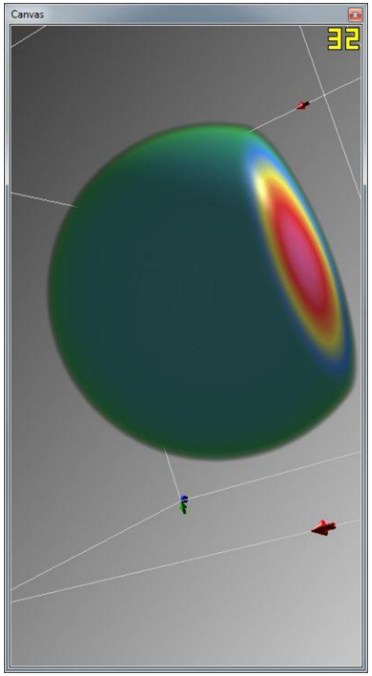


Texturing

- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
  - simulating materials (wood, bricks, even mirrors)
  - reducing geometric complexity

Visual detail is in the image,
not in the geometry!



Working with Textures

1. Specify a texture
   - Read or generate an image
   - Assign image to texture
   - Bind texture
   - Enable texturing
2. Specify texture parameters (wrapping, filtering)
3. Assign texture coordinates to vertices.
   As with colors, OpenGL interpolates the texture inside geometric objects.

Texture Objects

- Like display lists for texture images
  - one image per texture object
  - stored on the GPU
  - may be reused by several graphics objects

- Generate texture names

```
glGenTextures (n, *texIds);
```

- Bind textures before using

```
glBindTexture (target, texId);
```

- Delete textures when you are done

```
glDeleteTextures(n, *texIds);
```

Specify Texture Image

- Define a texture image from an array of texels in CPU memory

```
glTexImage2D (target, level, components,
   w, h, border, format, type, *texels);
```

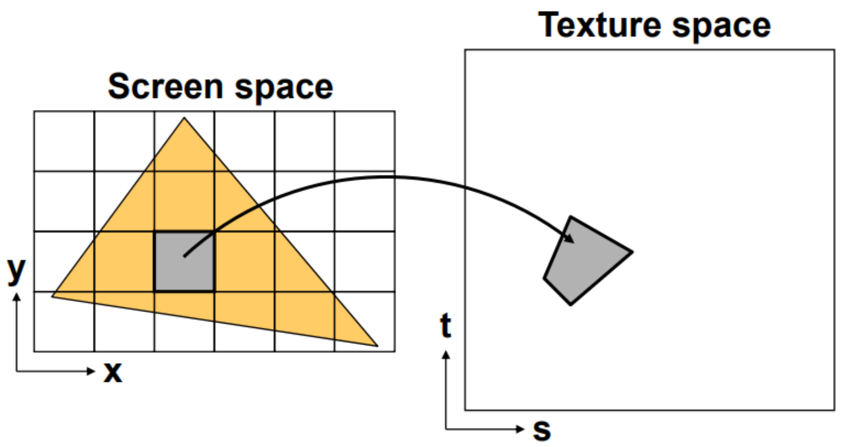
  - Dimensions w, h should be powers of 2
  - If not, use `gluScaleImage`

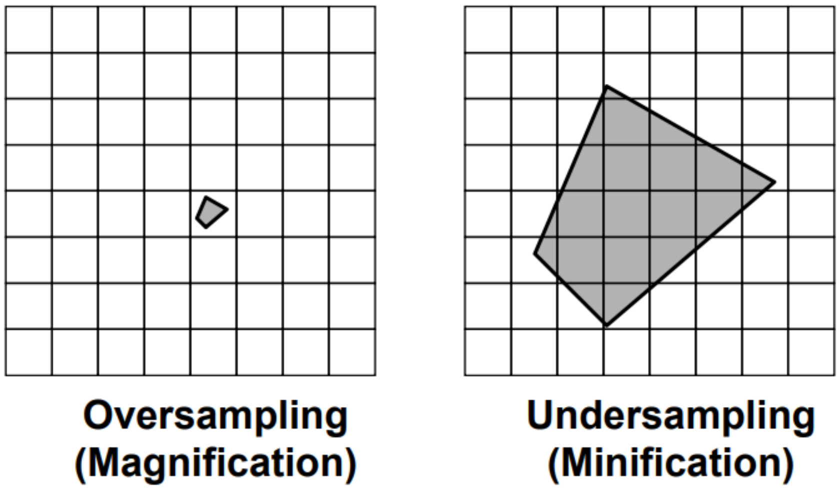
Texture Properties

- Filter Modes
  - minification / magnification
  - special mipmap minification filters
- Wrap Modes
  - clamping or repeating
- Environment parameters
  - how to mix primitive color with texture color
    - blend, modulate or replace texels

Filter Modes

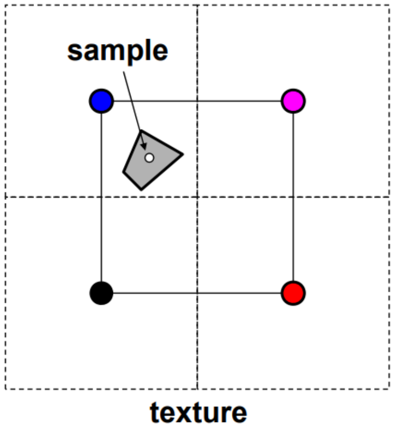
- Sampling density in texture space rarely matches the sample density of the texture itself.
- Texture mapping is subject to aliasing errors that can be controlled through filtering.
- Filter modes control how pixels are minified or magnified.

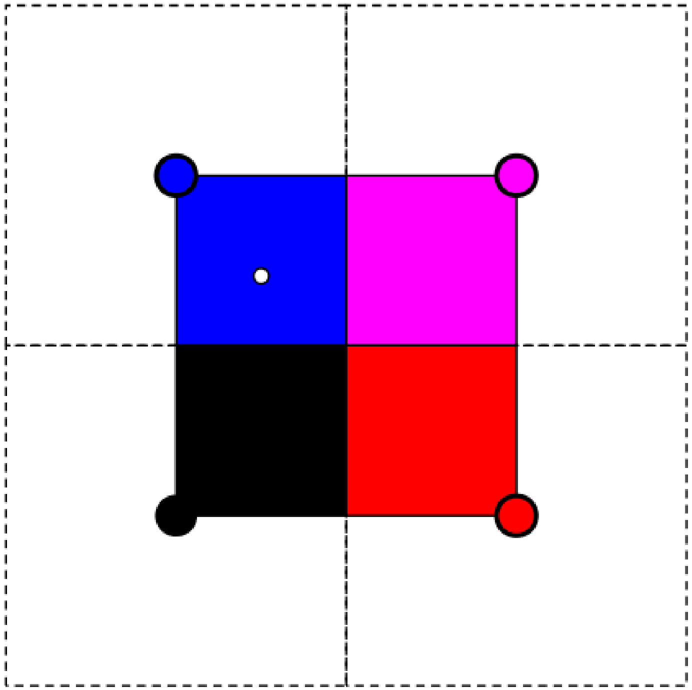




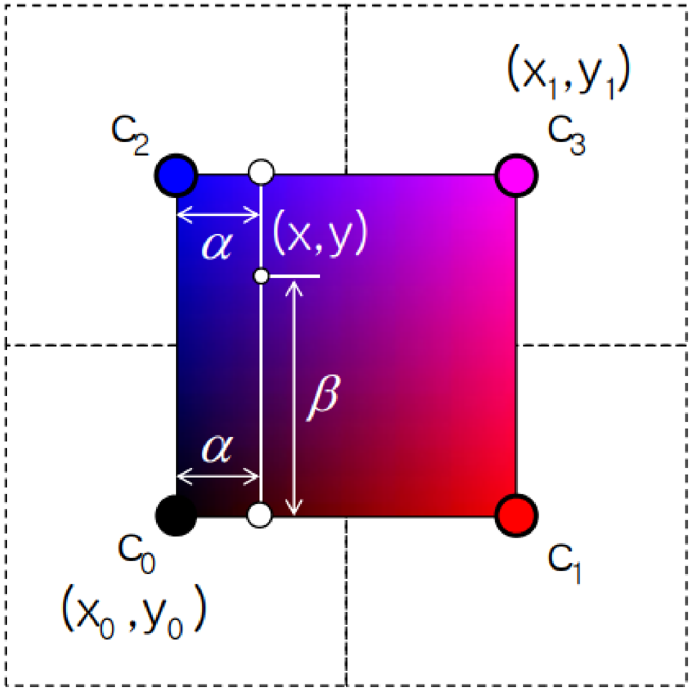

```
glTexParameteri (target, type, mode);
```

Oversampling





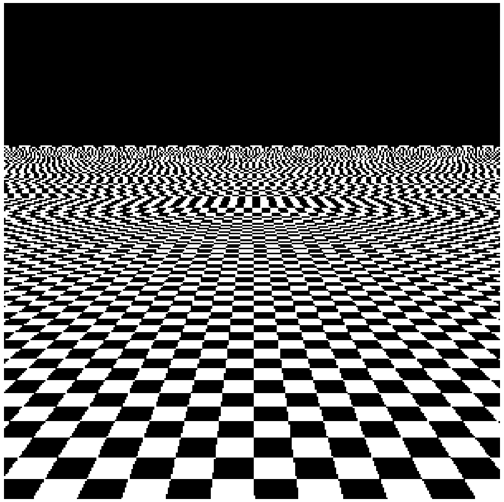
GL_NEAREST



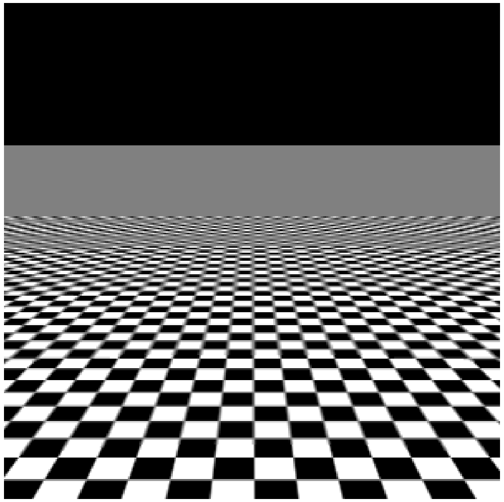
GL_LINEAR

Undersampling (Mipmapping)

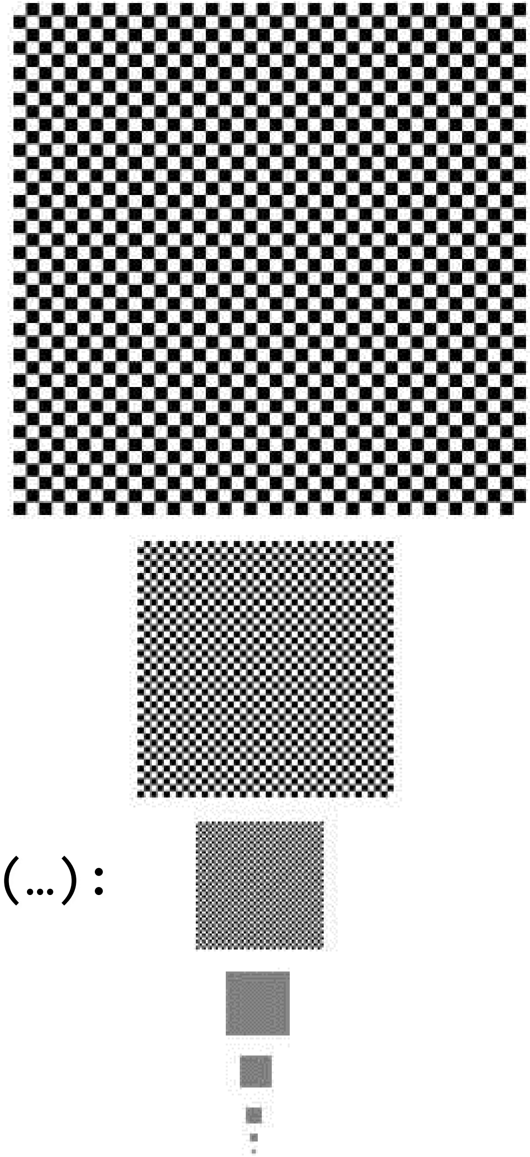
- Prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors (“shimmering”)



No mipmapping



mipmapping



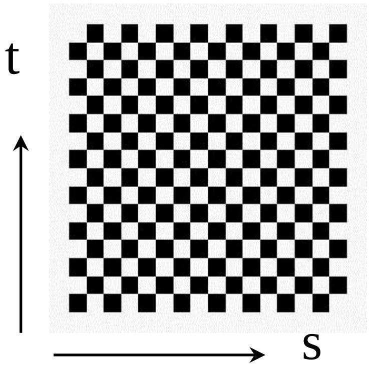
- GLU mipmap builder routines: `gluBuild*DMipmaps(…):`

```
gluBuild2DMipmaps
(GL_TEXTURE_2D, 3, width, height,
 GL_RGB, GL_UNSIGNED_BYTE, data);
```

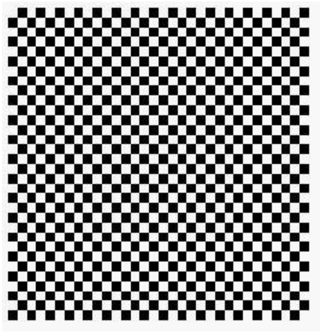
Wrap Mode

Determines what should happen if a texture coordinate lies outside of the [0,1] range.

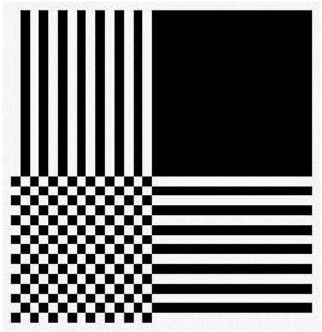
```
glTexParameteri (GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri (GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_REPEAT)
```



texture



GL_REPEAT wrapping



GL_CLAMP wrapping

Environment parameters

- Controls how the texture is applied

```
glTexEnv{fi}[v] (GL_TEXTURE_ENV, prop, param)
```

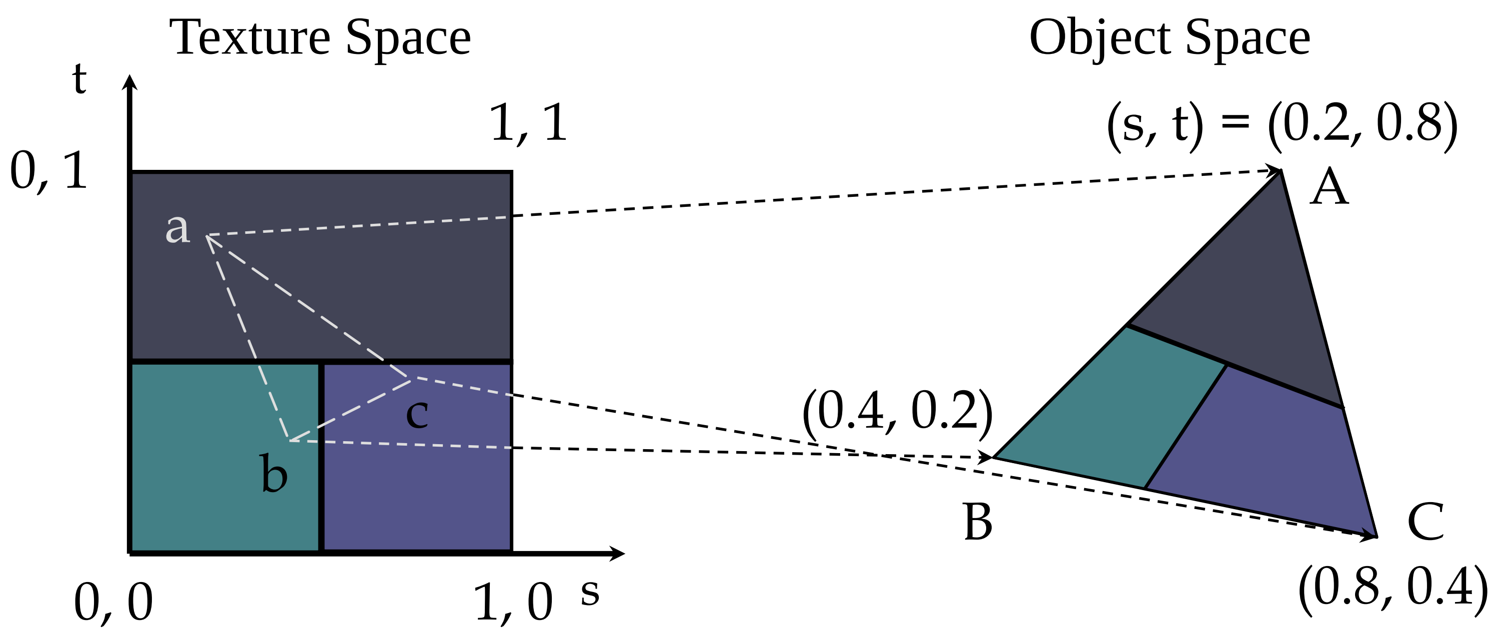
- Most interesting property: GL_TEXTURE_ENV_MODE
  - GL_MODULATE - multiply texel and fragment color
  - GL_BLEND - linearly blend texel, fragment, env color
  - GL_REPLACE - replace fragment’s color with texel

Example

```
glTexEnvf(
 GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Texture Mapping

- Based on texture coordinates
- `glTexCoord*()` specified at each vertex



Procedural Texture (example)

```
GLuint createTexureChecker()
{
        GLuint texture;
        int i, j, c;
        for (i = 0; i < checkerSize; i++)
                for (j = 0; j < checkerSize; j++) {
                        c = ((((i & 0x8) == 0) ^ ((j & 0x8)) == 0)) * 255;
                        checkerImage[i][j][0] = (GLubyte)c;
                        checkerImage[i][j][1] = (GLubyte)c;
                        checkerImage[i][j][2] = (GLubyte)c;
                        checkerImage[i][j][3] = (GLubyte)255;
                }
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                checkerSize, checkerSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, checkerImage);
        return texture;
}
```

Load Texture (from bmp)

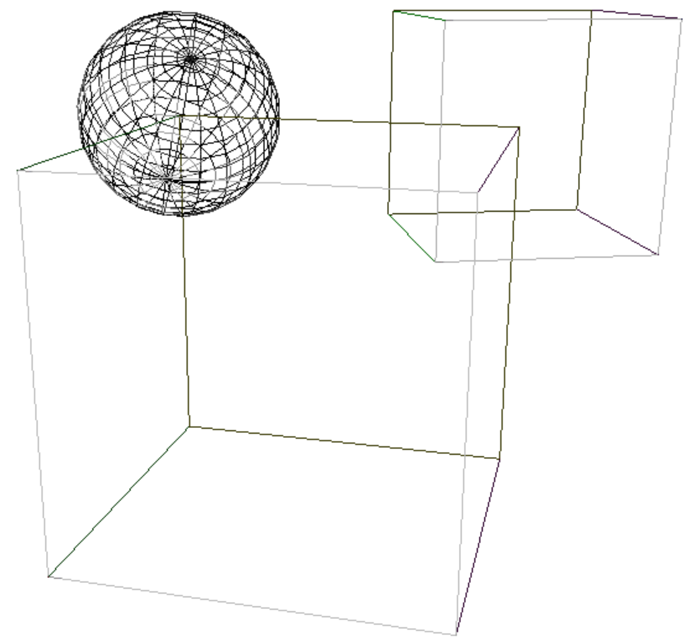
```
GLuint loadTextureFromBMP(const char * filename, int width, int height) {
	GLuint texture = 0; unsigned char * data;
	FILE * file; errno_t err;
	if ((err = fopen_s(&file, filename, "rb")) != 0) printf("Error: Texture was not opened.\n");
	else {
		data = (unsigned char *)malloc(width * height * 3);
		fread(data, width * height * 3, 1, file);
		fclose(file);
		for (int i = 0; i < width * height; ++i) { // bmp files are encoded BGR and not RGB
			int index = i * 3; unsigned char B, R;
			B = data[index]; R = data[index + 2];
			data[index] = R; data[index + 2] = B;
		}
		glGenTextures(1, &texture);
		glBindTexture(GL_TEXTURE_2D, texture);
		glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
		glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
		glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
		glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
		glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
		gluBuild2DMipmaps(GL_TEXTURE_2D, 3, width, height, GL_RGB, GL_UNSIGNED_BYTE, data);
		free(data);
	}
	return texture;
}
```

Exercise

- Create two textures
  - Define one procedurally
  - Load one from a file
- Apply one texture to a side of the cubes and the other to the sphere

Add-on I: Wireframe

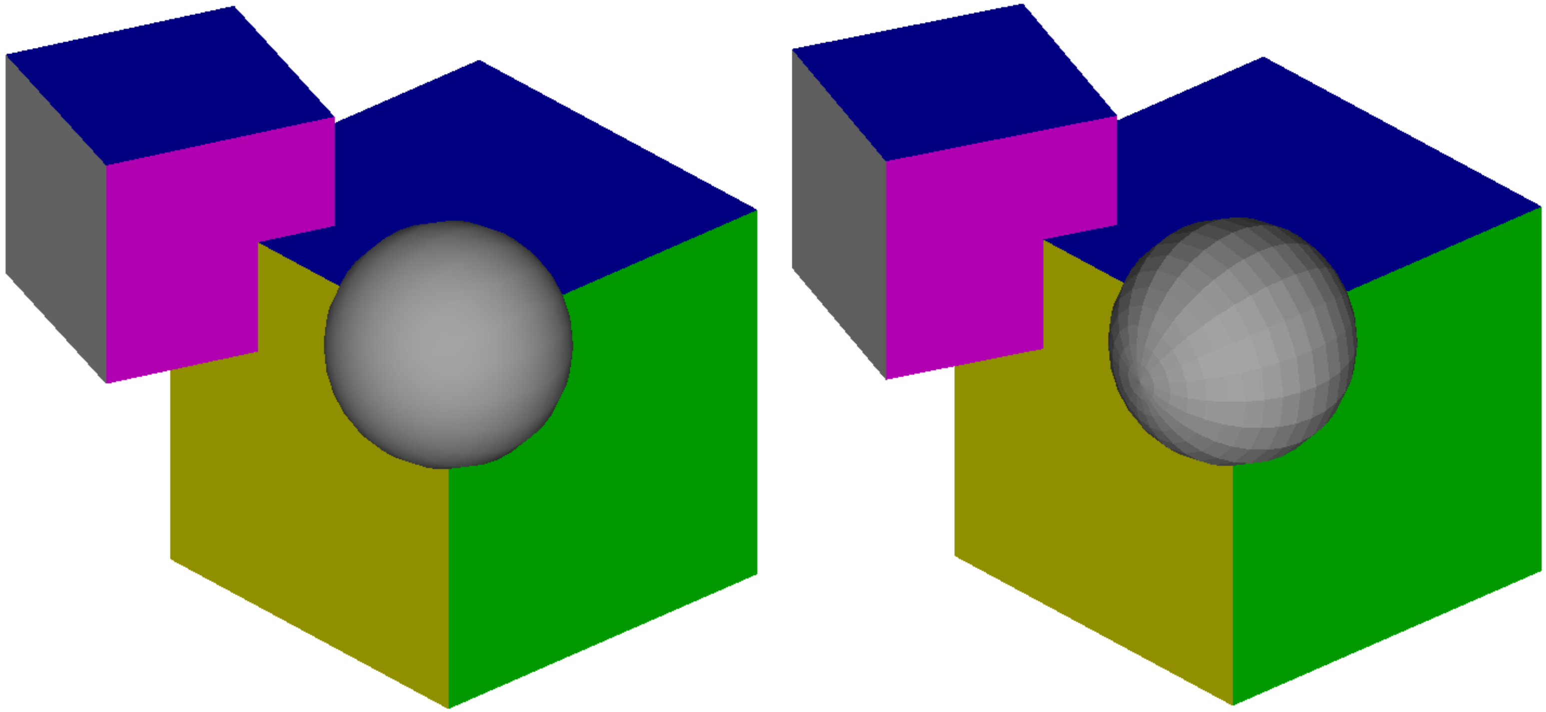
```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```



Exercise: Add wireframe mode to the scene

Add-on II: Flat Shading

```
glShadeModel(GL_SMOOTH);
glShadeModel(GL_FLAT);
```



Exercise: Add flat shading to the scene

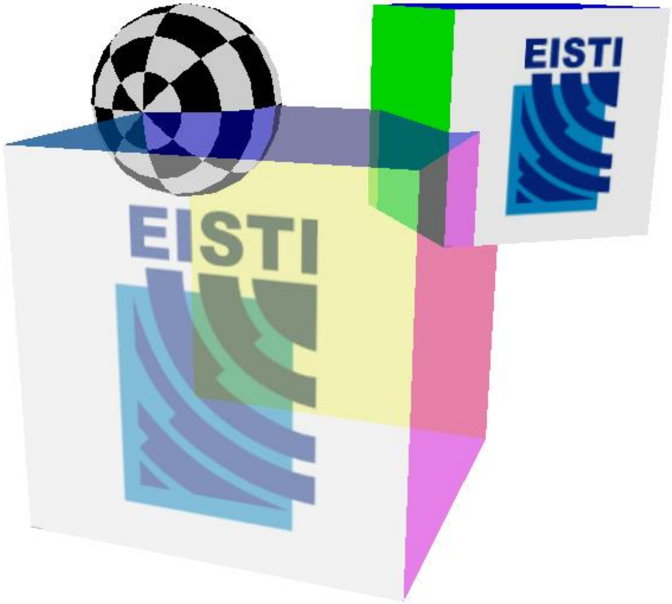
Add-on III: Alpha blending

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 arise only when you try to render one translucent polygon behind another. This is a difficult sorting problem.

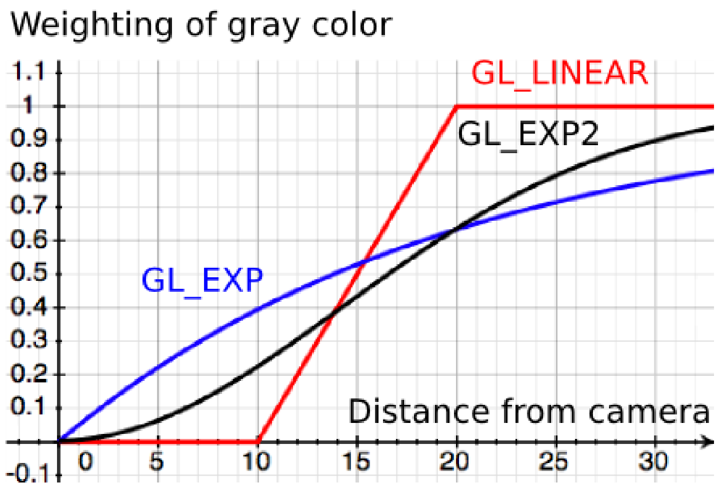
```
glColor4f(1.0, 1.0, 0.0, 0.5f);
// 4th parameter is alpha
...
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
   GL_ONE_MINUS_SRC_ALPHA);
```

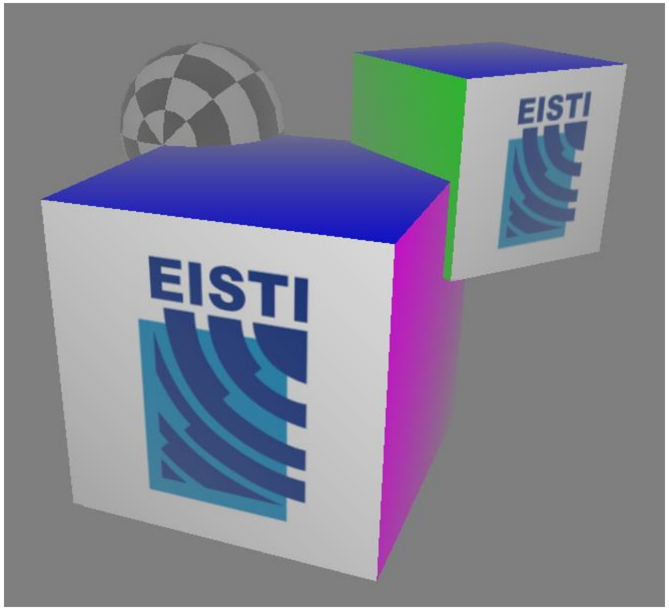


Exercise: Add translucency to the maincube

Add-on IV: 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

(dont forget to change the background color)

Your stunning OpenGL1 demo

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