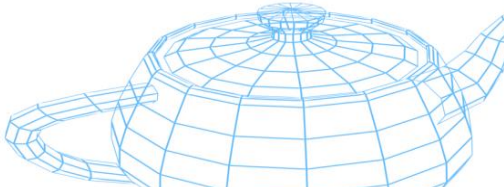
SUPSI

Computer Graphics

OpenGL (1): FreeGLUT, contexts, buffers and first steps

Achille Peternier, lecturer

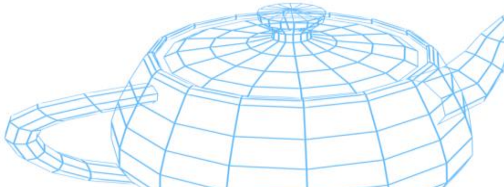


FreeGLUT

- **Free** alternative to the Open**GL U**tility **T**oolkit.

- Evolution/clone of GLUT, originally created by Mark Kilgard (Nvidia) and abandoned in 1999:
  - Less restrictive license.

- FreeGLUT started in 1999 by Pawel Olzsta:
  - Open-source.
  - No code inherited from GLUT.



FreeGLUT

- Event-based (callbacks).
- Supports user-input through mouse and keyboard.
- Provides a menu/sub-menu system.
- Supports printing text to the OpenGL window.
- Provides a series of built-in, dynamically generated 3D objects.
- Available on Windows, Linux, MacOS, etc.

FreeGLUT

- Windows:
  - Download and compile the project:
    - http://freeglut.sourceforge.net/
  - Use the .lib + .dll or .lib-only (static) version.
  - Define **`FREEGLUT_STATIC`** for static linking:
    - The static lib is included (and used) in the tutorials and series’ solutions.

- Ubuntu:
  - Execute: **`sudo apt install freeglut3-dev`**
  - Link to *libglut.so* (dynamic) or *libglut.a* (static).

FreeGLUT

- Typical usage:
  - Initialize the library.
  - Create one or more windows for graphic output.
  - Create menus (optional).
  - Register callback functions.
  - Enter the main loop.

FreeGLUT

- **`void glutInit(int *argc, char *argv[]);`**
  - Accepts -display, -geometry

- **`void glutInitDisplayMode(flags);`**
  - Accepts:
    - **`GLUT_SINGLE`**, **`GLUT_DOUBLE`** single/double buffering
    - **`GLUT_RGB`**, **`GLUT_RGBA`** color mode
    - **`GLUT_DEPTH`** enables the z buffer

    ...
  - E.g.: **`glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA);`**

FreeGLUT

- **`void glutInitWindowSize(int width, int height);`**

- **`void glutInitWindowPosition(int x, int y);`**

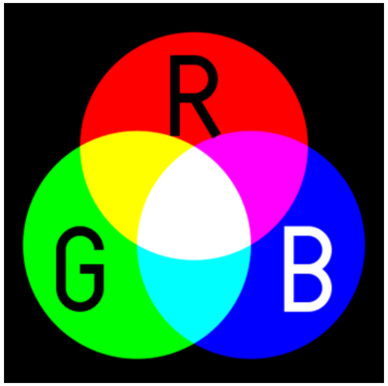
- **`int glutCreateWindow(char *name);`**
  - name = window title.
  - returns the window ID.

- ```
  int glutCreateSubWindow(int parent_window, int x, int y,
                          int width, int height);
  ```
  ```

FreeGLUT

- Callback registration:
  - **`glutDisplayFunc(func. ptr.);`**
    - Invoked each time the output scene must be re-rendered.
    - Call **`glutPostWindowRedisplay(winId)`** to force a refresh.
  - **`glutReshapeFunc(func. ptr.);`**
    - Triggered each time the window size is changed.
  - **`glutMouseFunc(func. ptr.);`**
    - Triggered each time the mouse is used.
  - **`glutKeyboard(func. ptr.);`**
    - Triggered each time a keyboard key is pressed.
  - **`glutSpecial(func. ptr.);`**
    - Same as before, but for special keyboard keys such as the arrows.


Tutorial
Basic FreeGLUT
VIDEO
DISPLAY

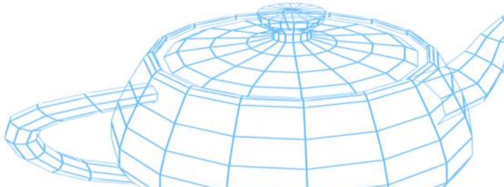


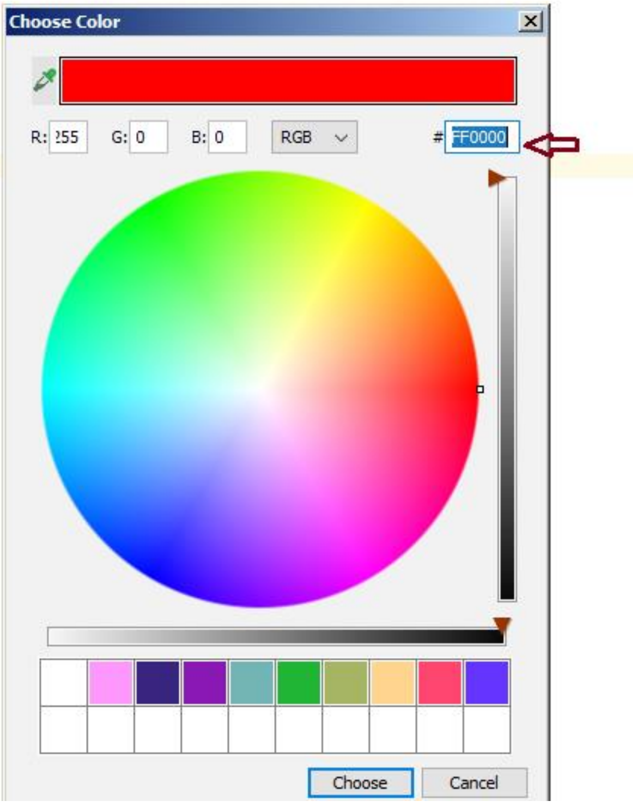
RGB

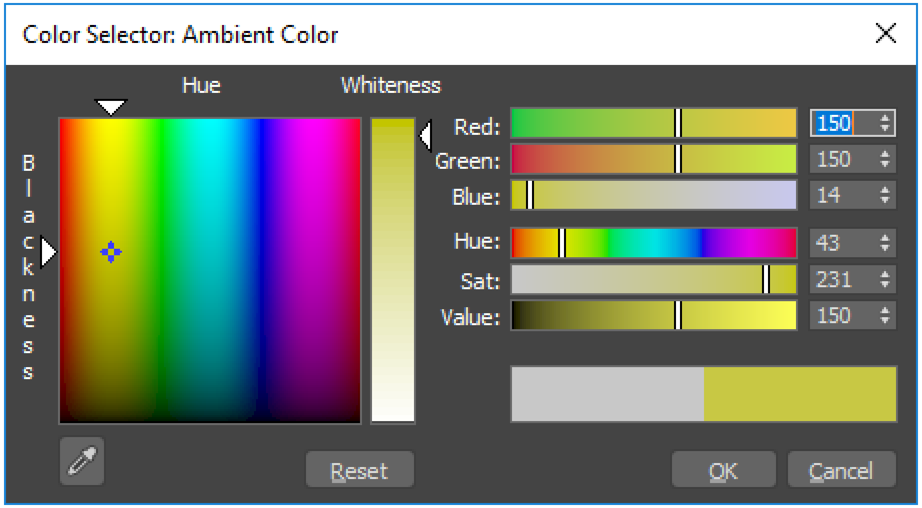
- **R**ed **G**reen **B**lue :
  - Color model used to express colors based on the intensity of the three base colors.
  - Works for light-emitting sources (and not for painting).
  - Additive method:

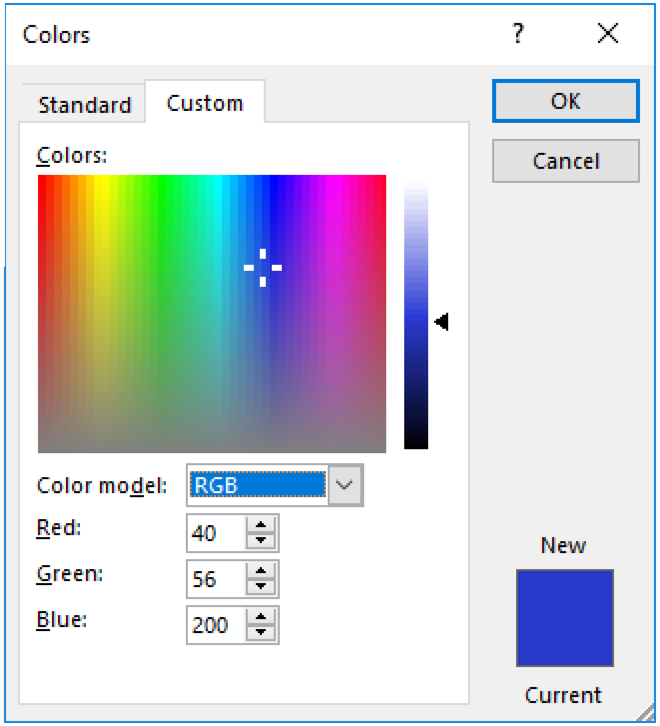
$$\text{color} = R_{\text{intensity}} + G_{\text{intensity}} + B_{\text{intensity}}$$

  - Several ways to encode values:
    - Bytes [0-255], e.g.: [255 0 0], [0 255 0], [128 128 128]
    - Float [0.0-1.0], e.g.: [1.0 0.0 0.0], [0.0 1.0 0.0], [0.5 0.5 0.5]
    - Hexadecimal [00-FF], e.g.: #FF0000, #00FF00, #7F7F7F

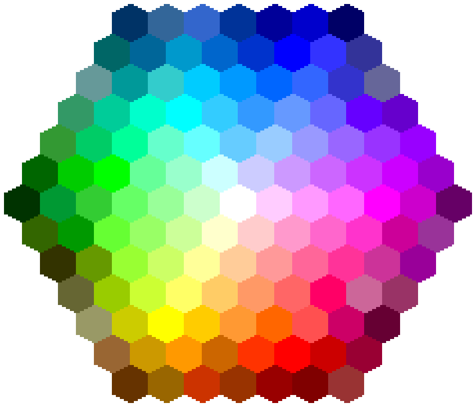


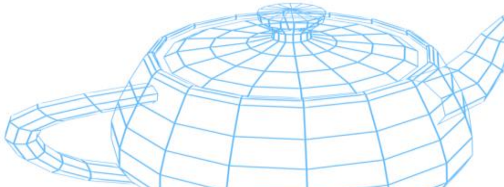

Choose Color
R: 255
G: 0
B: 0
RGB
FF0000
Choose
Cancel


Color Selector: Ambient Color
Hue
Whiteness
Blackness
Red: 150
Green: 150
Blue: 14
Hue: 43
Sat: 231
Value: 150
Reset
OK
Cancel


Colors
Standard
Custom
OK
Cancel
Colors:
Color model: RGB
Red: 40
Green: 56
Blue: 200
New
Current


#f2d91c



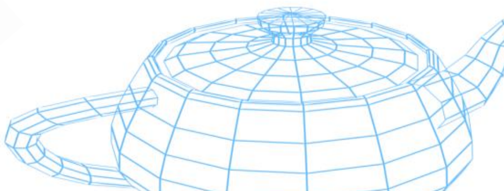


RGBA

- Another channel (alpha) is added for storing additional information (**usually** transparency):
  - As already seen for the intensities of the other channels, the alpha channel intensity is defined as *0 = transparent*, and *max value = completely solid.*

- RGBA using float = 4 * sizeof(float) = 16 bytes = 128 bit:
  - Good for memory alignment.
  - Perfect for SIMD 128 bit registers.

- Specifying transparent alpha values **will not** automatically activate transparency in OpenGL!
  - Transparency is a much more complex topic that we will see later.



RGBA

- No need to create a new class: RGBA = XYZW.
  - Reuse **`glm::vec3`**, e.g.:

```
// Define red [1.0 0.0 0.0]:
glm::vec3 color;
   color.r = 1.0f;
   color.g = 0.0f;
   color.b = 0.0f;
```

  - …or **`glm::vec4`** for RGBA colors, e.g.:

```
// Define red with alpha channel [1.0 0.0 0.0 1.0]:
glm::vec4 color;
   color.r = 1.0f;
   color.g = 0.0f;
   color.b = 0.0f;
   color.a = 1.0f;
```

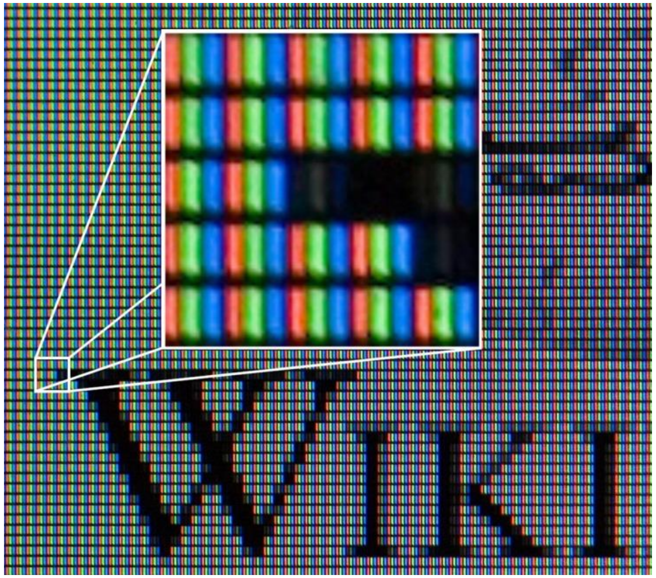
Main buffers

- A rendering-context is initialized by specifying the characteristics of these four buffers:
  - **Framebuffer** = main output buffer (where you render the final image).
  - **Back buffer** = the hidden twin of the framebuffer, where the next frame is being rendered [almost mandatory].
  - **Depth buffer** (or **Z buffer**) = stores Z values for each pixel of the framebuffer [optional].
  - **Stencil buffer** = additional buffer for per-pixel logic operations [really optional].
  - **Accumulation buffer(s)** = one or more additional buffers for storing a series of images [extremely optional].

Framebuffer

- Memory segment containing the image information to display through the graphics device.

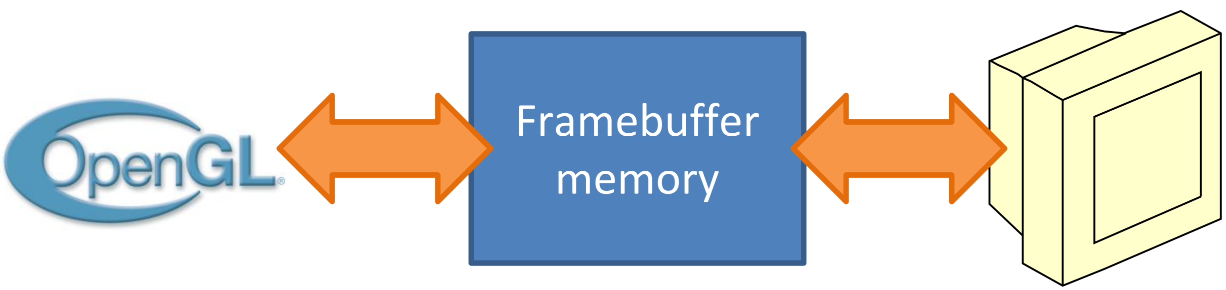
- Contains pixel colors:
  - RGB and RGBA on modern displays.
  - Single bits for older, monochromatic monitors.



Framebuffer

- The memory information stored in the framebuffer is accessed by the device to periodically refresh the pixels rendered on the screen:
  - Typical refresh-rate between 60 and 120 Hz:
    - 120 Hz useful for stereographic rendering (60 Hz per eye).

- To avoid visual artifacts (screen flickering and tearing), video memory refreshing is synchronized with the screen refresh rate.




Tear Point #1 --->
Tear Point #2 --->

Framebuffer

- Vertical synchronization:
  - Old terminology used on CRT monitors:
    - The graphics card waits for the vertical beam to reach the lower-right corner of the screen.
    - While the beam is reset to its upper-left position, the graphics memory is refreshed.
    - Additional modifications are prevented until the next reset.
  - It is used to synchronize the rendering speed with the monitor refresh frequency.
- Double buffering:
  - Uses two buffers: front (main one, rendered on the screen) and back.
  - Instead of rendering to the screen memory buffer, a secondary (hidden) buffer is used.
  - Once the image is ready on the back-buffer, it is copied to the front-buffer (usually during vertical sync):
    - As optimization, back- and front-buffer pointers are swapped (zero copy, page flipping/ping-pong buffering).
- Triple buffering:
  - Same as double buffering, but two back-buffers are used.
  - With double buffering, when the back-buffer image is ready, the pipeline is stalled waiting for the front-buffer to be available:
    - With triple buffering, there's always a non-locked buffer for rendering.

Framebuffer

- New technologies for getting rid of the GPU-display synchronization problem:
  - NVidia G-Sync: dedicated HW embedded in monitors to dynamically sync with the GPU signal.
  - AMD FreeSync: a similar mechanism but working without dedicated components and released patent-free.

Framebuffer

- Different framebuffer color depths:
  - RGB:
    - 1 bit = monochromatic.
    - 4 bit = 16 colors (from a palette).
    - 8 bit = 256 colors (from a palette).
  - RGB/RGBA:
    - 15 bit = high-color (2^{15} colors):
      - 5+5+5 (RGB).
    - 16 bit = high-color (2^{16} colors):
      - 5+5+5+1 (RGBA) = 15 bit high-color + alpha channel.
      - 5+6+5 (RGB) = 16 bit high-color:
        - » Human eye is more sensitive to green.
      - 4+4+4+4 (RGBA).
    - 24 bit = true-color (2^{24} colors, ~16 millions):
      - Human eye can recognize up to 10 million colors.
  - RGBA:
    - 32 bit = true-color (24 bit) + 8 bit alpha channel.
  - > 32 bit:
    - High Dynamic Range (HDR), professional devices, intermediate steps.

Framebuffer

- RGB or RGBA mode must be specified during the context creation:
  - **`glutInitDisplayMode(GLUT_RGB or GLUT_RGBA);`**

- Double buffering must be specified during the context creation:
  - **`glutInitDisplayMode(... | GLUT_DOUBLE);`**
  - Once finished with the rendering of the current frame, front-/back-buffer swapping must be explicitly invoked:
    - **`glutSwapBuffers();`**

Framebuffer

- Framebuffer clear color is specified through its RGBA components:
  - **`glClearColor(red, green, blue, alpha);`**
    - Alpha is required in RGB mode, too.

- Framebuffer is cleared explicitly through:
  - **`glClear(GL_COLOR_BUFFER_BIT);`**

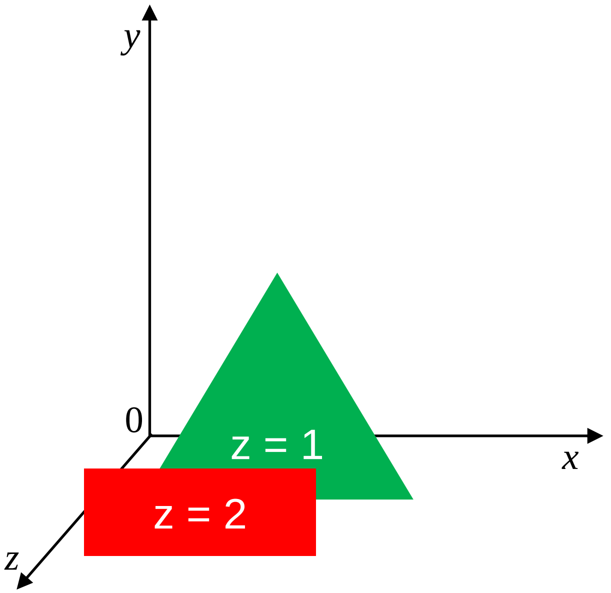
- Details about the current context are retrieved through:
  - **`glutGet(enum);`**
    - E.g.: **`GLUT_WINDOW_BUFFER_SIZE`**, **`GLUT_WINDOW_RED_SIZE`**, **`GLUT_WINDOW_GREEN_SIZE`**, **`GLUT_WINDOW_DOUBLEBUFFER`**, …

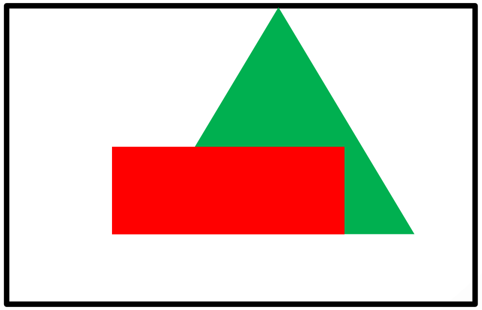
Z buffer

- Is used to store the depth (z) value of each pixel of the framebuffer.
- Works like a “sonar”.
- Operations to the framebuffer are conditioned by the z buffer current state.

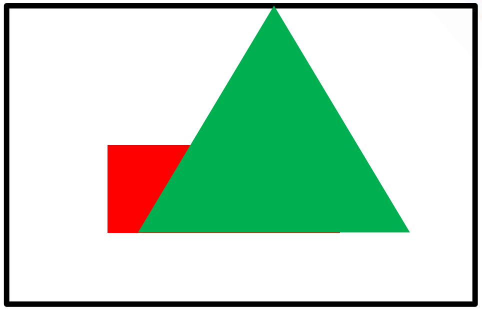
Z buffer

- Without depth test:





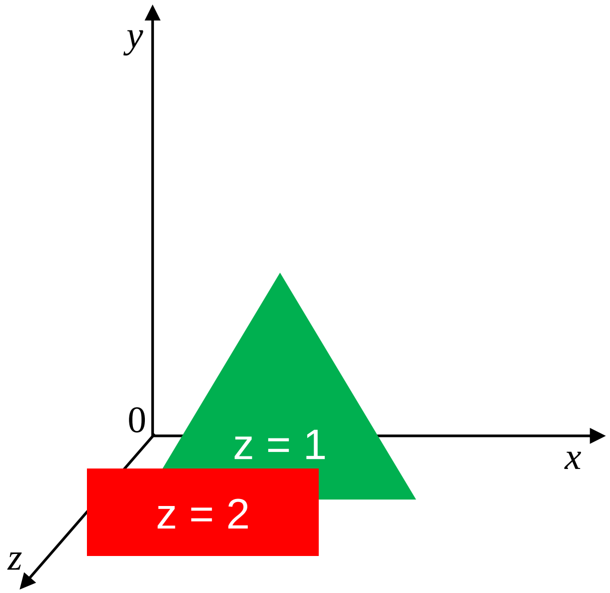
a) first the triangle,
then the rectangle

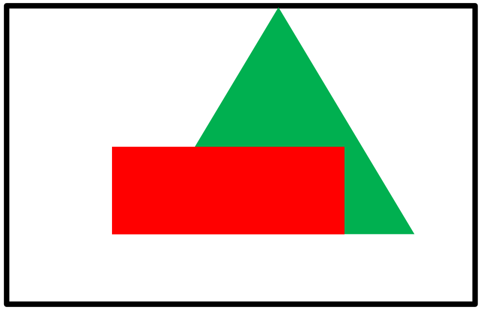


b) first the rectangle,
then the triangle

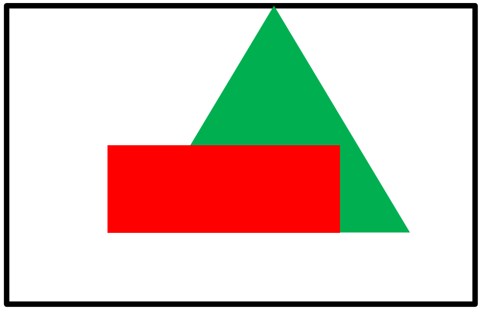
Z buffer

- With depth test: order-independent rendering



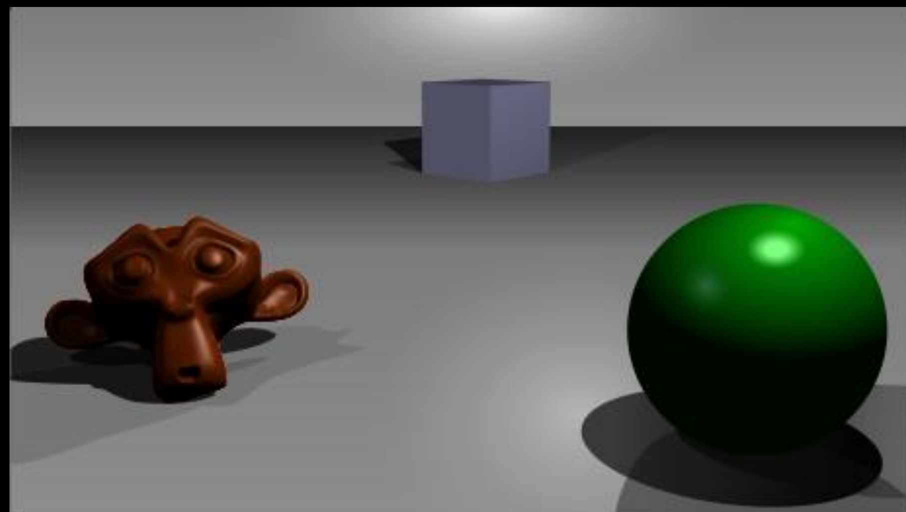


a) first the triangle, then the rectangle



b) first the rectangle, then the triangle

Z buffer



A simple three-dimensional scene

Z-buffer representation

Z buffer

- Contains the perpendicular distance of each pixel of the framebuffer relative to the near clipping plane.

- Values in normalized device coordinates $[-1 < z < 1]$ are resampled into the $[0 < z < 1]$ range.

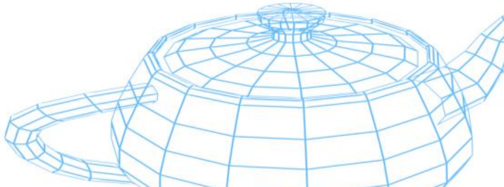
- Accuracy depends on the number of bit used:
  - Typically 16/24 bit.
  - The 24 bit z buffer is often padded with an 8 bit stencil buffer to reach 32 bit boundaries.
  - 32 bit z buffers are possible only through off-screen framebuffers (via modern OpenGL framebuffer objects).

Z buffer

- Precision is higher for objects closer to the *zNear* plane.

- In general, try to push the *zNear* plane out and the *zFar* plane in as much as possible:
  - Your z buffer accuracy corresponds to the discretization of the space *zFar - zNear* using X bit, where X is your z buffer bit depth.

- **Always** keep *zNear* > 0.



Z buffer

- Must be specified during the context creation:
  - **`glutInitDisplayMode(... | GLUT_DEPTH);`**

- Must be explicitly enabled:
  - **`glEnable(GL_DEPTH_TEST);`**

- Z buffer must be cleaned before rendering:
  - **`glClear(… | GL_DEPTH_BUFFER_BIT);`**

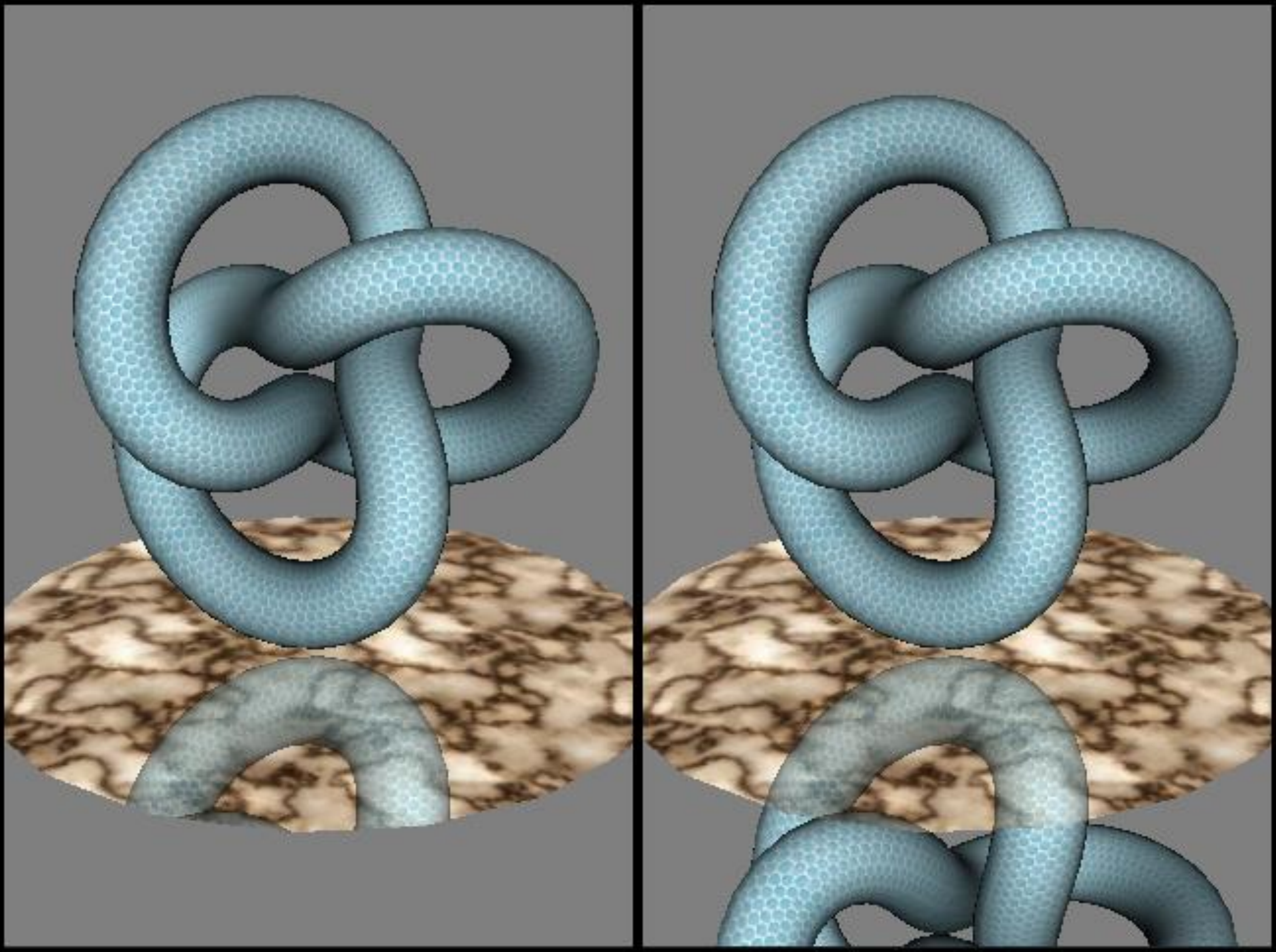
- Z buffer behavior must be configured:
  - **`glClearDepth(float);`** [default is 1.0]
  - **`glDepthFunc(enum);`** [default is **`GL_LESS`**]

Stencil buffer

- Optional buffer with the same dimension of the color and z buffers and a typical depth of 8 bit:
  - Could be just 1 bit.
  - Interaction with the z buffer:
    - Stencil buffer values can be modified according to the result of the depth test.

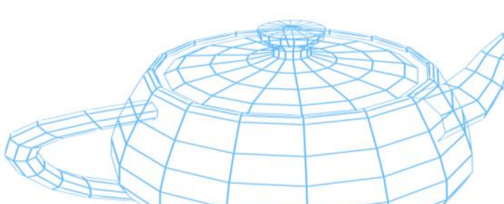
- Mainly used to limit the rendering to specific, pixel-precise areas:
  - Planar reflections.

- Other advanced applications involve volume shadows, constructive solid geometry, portals, etc.



Stencil buffer

- Must be specified during the context creation:
  - `glutInitDisplayMode(... | GLUT_STENCIL);`
- Must be explicitly enabled:
  - `glEnable(GL_STENCIL_TEST);`
- Stencil buffer must be cleaned before rendering:
  - `glClear(… | GL_STENCIL_BUFFER_BIT);`
- Stencil buffer’s behavior must be configured:
  - `glClearStencil(int);` [default is 0]
  - `glStencilFunc(enum, ref, mask);`
  - `glStencilOp(fail, zfail, zpass);`



Main buffers

OpenGL context 1 (640x480, no double buffer):

Framebuffer (24 bit) + Z buffer (24 bit) + Stencil buffer (8 bit)

640x480 = 307’200 pixels
24+24+8 = 56 bit
~2 MB of VRAM
(48 MB/s @ 24 fps)

OpenGL context 2 (1920x1080, double buffer):

Framebuffer (32 bit) + Z buffer (24 bit)

1920x1080 = 2’073’600 pixels
32+24 = 56 bit
~14 MB of VRAM (~20 MB x2)
(336 MB/s @ 24 fps)

Per-vertex information

- Vertex position (as seen so far):
  - x, y, z[, w] (usually as *float*)

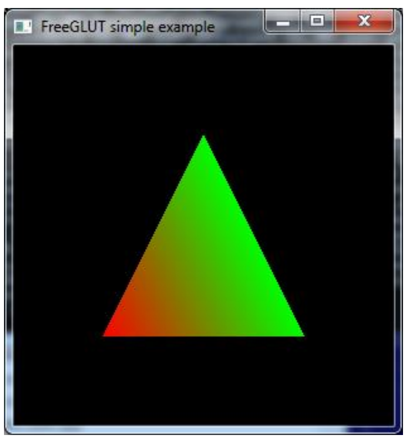
- Vertex color (RGB or RGBA):
  - r, g, b[, a] (usually as *byte*)

- …we will see additional per-vertex data later in the course (like normal vectors and texture coordinates).

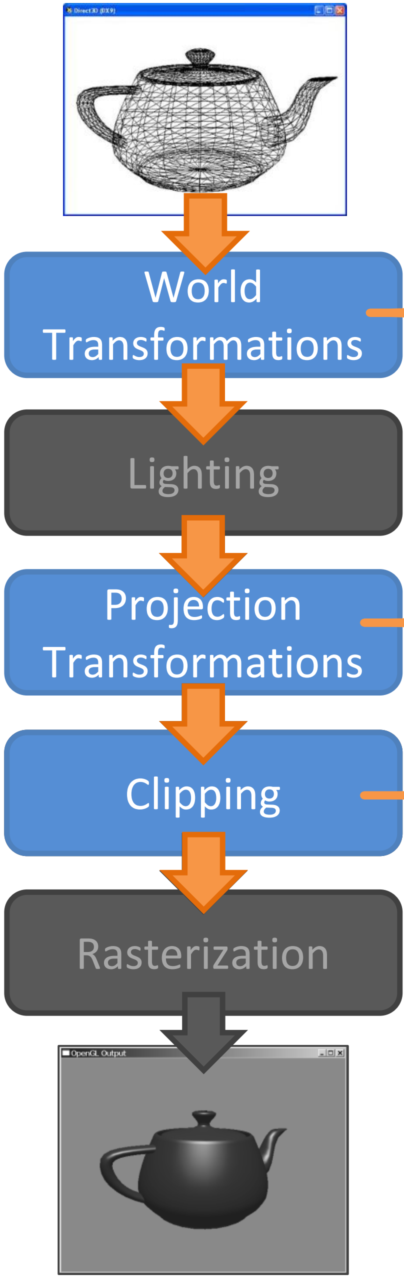
Immediate mode

- Generates primitives according to the type specified and the number of vertices passed.

- A new vertex is generated when **`glVertex*()`** is called, using the last color values specified.




```
glBegin(GL_TRIANGLES);
      glColor3f(1.0f, 0.0f, 0.0f);
         glVertex3f(0.0f, 0.0f, 0.0f);
      glColor3f(0.0f, 1.0f, 0.0f);
         glVertex3f(10.0f, 0.0f, 0.0f);
         glVertex3f(5.0f, 5.0f, 0.0f);
glEnd();
```



OpenGL Matrices

```
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

*(where m is typically cameraMat^{-1} * transMat * rotMat * scaleMat)*

```
glBegin(…)
   glVertex*(…)
glEnd()
```

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(m);
```

(where m is typically computed by glm::ortho or glm::perspective)

```
glViewport(startX, startY, width, height);
```

$$\begin{bmatrix} clip_x \\ clip_y \\ clip_z \\ clip_w \end{bmatrix} = \overbrace{\text{projMat}}^{\texttt{GL\_PROJECTION}} * \overbrace{\text{cameraMat}^{-1} * \text{transMat} * \text{rotMat} * \text{scaleMat}}^{\texttt{GL\_MODELVIEW}} * \overbrace{\begin{bmatrix} obj_x \\ obj_y \\ obj_z \\ 1 \end{bmatrix}}^{\texttt{glVertex*()}}$$

Each time you resize the window, don't forget to update the projection matrix and `glViewport()` values accordingly!

Matrices

- OpenGL stores, for each mode, the current matrix.
- Current matrix is set by passing a **`glm::mat4`** pointer to the method **`glLoadMatrixf(float *);`**

```
#include <glm/gtc/type_ptr.hpp>

glMatrixMode(GL_MODELVIEW);
glm::mat4 mv = cameraInv * translation * rotation;
glLoadMatrixf(glm::value_ptr(mv));

glMatrixMode(GL_PROJECTION);
glm::mat4 pj = glm::perspective(…)
glLoadMatrixf(glm::value_ptr(pj));
```

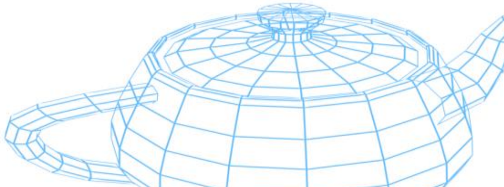
Matrices

- For any given matrix used to position an object in world coordinates:

$$\mathbf{M} = \begin{array}{c} \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ \begin{array}{cccc} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{t} \end{array} \end{array}$$

where **x**, **y**, **z** and **t** are column vectors representing the object local:

- Right direction (**x**).
- Up direction (**y**).
- Forward direction (**z**).
- Position (**t**).



LookAt

- Commodity method for computing the camera matrix from a set of given parameters (eye, center and up):
  - **eye** (vec3): is the position of the camera.
  - **center** (vec3): is the position of the point the camera is looking at.
  - **up** (vec3): is a vector indicating the orientation of the world (typically 0, 1, 0).

- Available through the **`glm::lookAt()`** method:

```
glm::vec3 eye = glm::vec3(0.0f, 0.0f, 10.0f);
glm::vec3 center = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);

glm::mat4 viewMat = glm::lookAt(eye, center, up);
```

Camera demo

$$\textbf{Perspective} = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar + zNear}{zNear - zFar} & \frac{2 \times zFar \times zNear}{zNear - zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$


Tutorial
Advanced FreeGLUT
VIDEO
DISPLAY