

# CENG 477

# Introduction to Computer Graphics

## Graphics Hardware and OpenGL

# Introduction

- Until now, we focused on graphic **algorithms** rather than hardware and implementation details
- But graphics, without using specialized tools and/or hardware would simply be too slow for most applications
- We will now learn about how a **GPU** works and how to program it using a specific API: **OpenGL**
- The presented ideas should apply to other APIs such as Direct3D with some modifications

# Graphics Hardware (GH)

- GH is a set of components which implements the forward rendering pipeline at a chip level called GPU
- Modern GPUs are programmable
- GPUs are massively parallel (orders of magnitude more parallel than CPUs)
- GPUs change continuously mainly due to the demands of the video game industry
- Big players:
  - AMD, Nvidia, Intel, Microsoft, Apple, Qualcomm, ...

# Graphics Processing Unit (GPU)

- How parallel are GPUs? Let's watch this demo:



# GPGPU

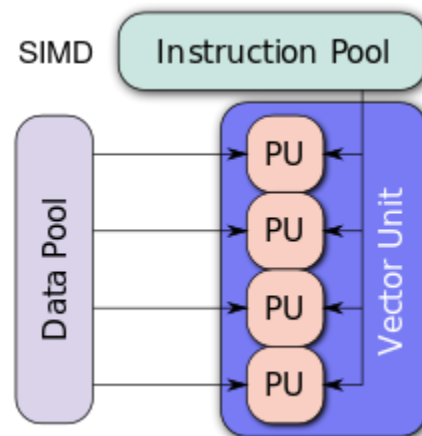
- As a result of this performance, GPUs are used in many tasks that are not related to graphics at all:
  - Called **GPGPU**: General-purpose computing on GPU
- Nvidia developed the **CUDA** language for GPGPU
- **OpenCL** is supported by AMD, Nvidia, and Intel
- Nowadays, many computational intensive tasks are performed on the GPU:
  - Image and video processing
  - Analyzing big data
  - Bioinformatics
  - Optimization
  - Machine learning, ...

# GPU Architecture

- GPUs are similar to CPUs in their building blocks (in fact they are somewhat simpler than CPUs):
  - Some logic to decode the instruction to be performed
  - Registers
  - Arithmetic logic units (ALUs)
  - Cache
  - Memory
- But they are massively parallel:
  - Data parallelism
  - Pipeline parallelism

# GPU Parallelism

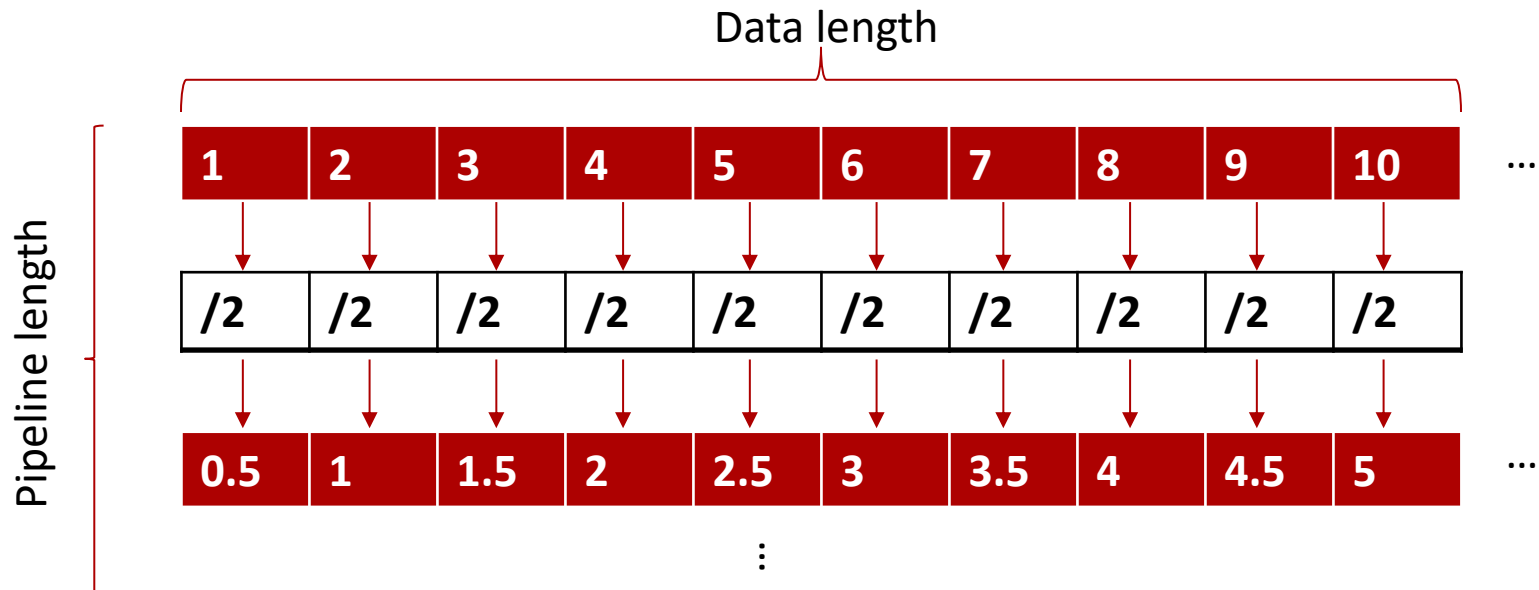
- What makes GPUs parallel?
- GPUs are SIMD architectures
  - **SIMD**: Single instruction multiple data
  - The same instruction is applied to thousands of data elements at the same time



wikipedia.com

# GPU Parallelism

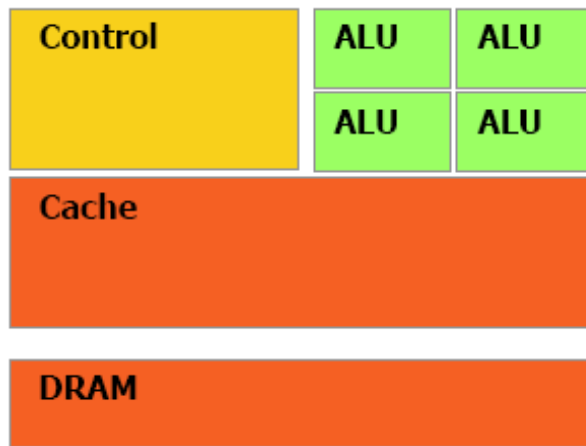
- This works well for **independent** tasks such as:
  - Transforming vertices
  - Computing shading for each fragment
- Ideal if the task is the same but the data is different:



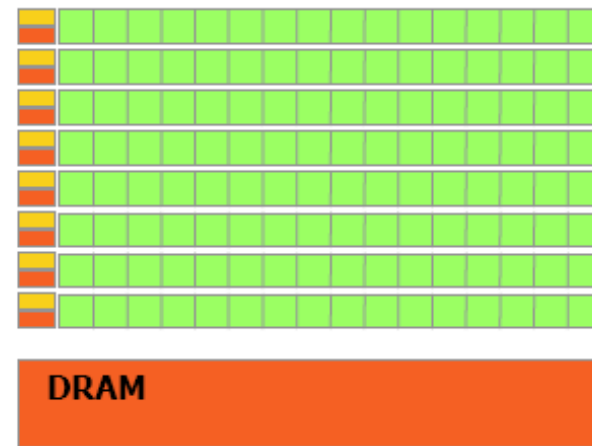


# GPU vs CPU

- GPUs have a larger number of ALUs allowing for data parallelism:



**CPU**



**GPU**

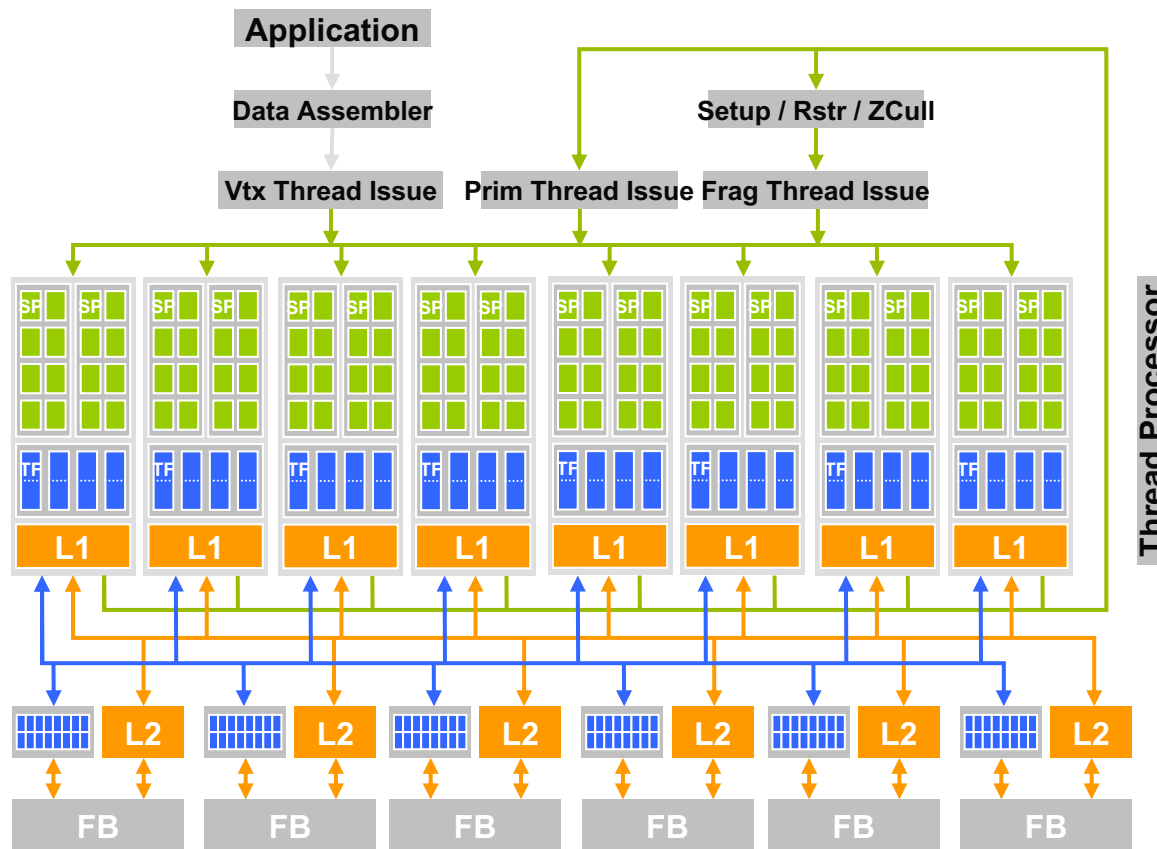
# GPU vs CPU

- Let's compare a good GPU with a good CPU

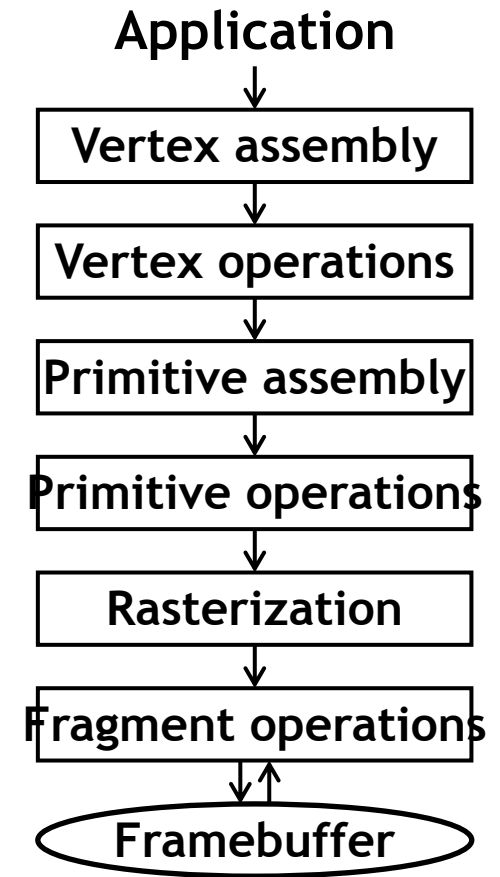
Intel i7-4790K	Nvidia GTX 1060
Cores: 4 (8 threads)	Cores: 1280
Clock: 4 – 4.4 GHz	Clock: 1.5 – 1.7 GHz
Power: 88W	Power: 120W
Memory BW: 25.6 GB/s	Memory BW: 192 GB/s



# Overall GPU Architecture

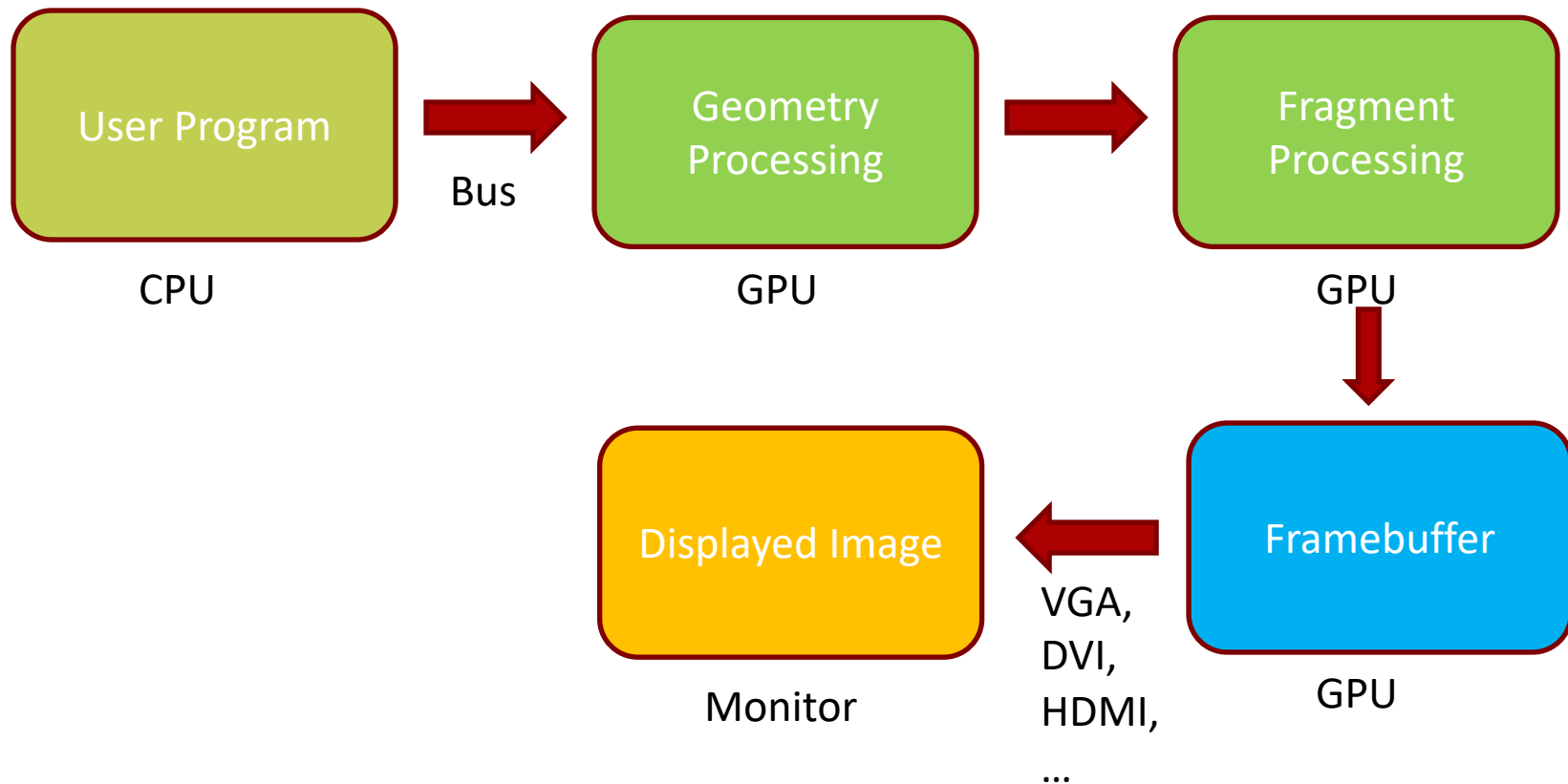


**NVIDIA GeForce 8800**



**OpenGL Pipeline**

# GPU Data Flow Model



# User Program

- The user program is an **OpenGL** (or Direct3D) program which itself runs on the CPU
- Also initially all data is in the main system memory
- The user program is responsible to arbitrate the overall flow and send data to GPU:
  - Open a window
  - Manage user interaction (mouse, keyboard, etc.)
  - Decide what to draw and when to draw
  - Ask GPU to compile shaders (programs to be later run on the GPU)

# Opening a Window

- Opening a window for rendering is not part of OpenGL
  - Each OS has a different mechanism
- There are some high-level APIs that simplify this process
  - Perhaps the simplest of these APIs is GLUT
  - You will learn GLFW in the recitation, which is simpler and better

```
glutInit(&argc, argv);  
glutInitDisplayMode(GLUT_RGBA |  
                    GLUT_DOUBLE |  
                    GLUT_DEPTH |  
                    GLUT_STENCIL);  
  
glutInitWindowPosition(100, 100);  
glutInitWindowSize(640, 480);  
glutCreateWindow("");
```

Buffers that  
we want

# Double Buffering

- Double buffering is a technique to avoid tearing
  - Problem happens when drawing and displaying the same buffer

With Tearing

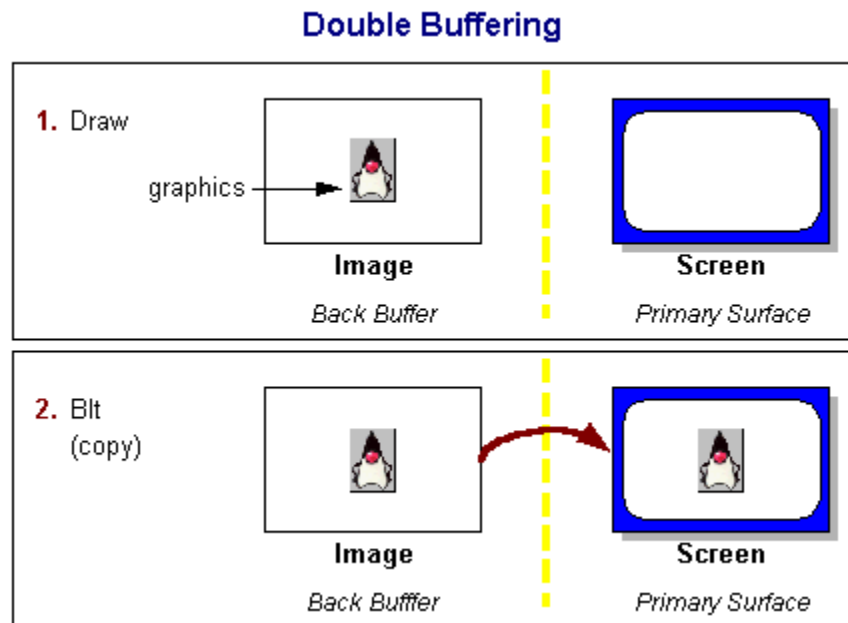


No Tearing



# Double Buffering

- To avoid such artifacts, we render to a **back buffer** and show that buffer only when drawing is complete (usually synchronized with monitor's refresh cycle)
  - Windowed mode requires a copy:

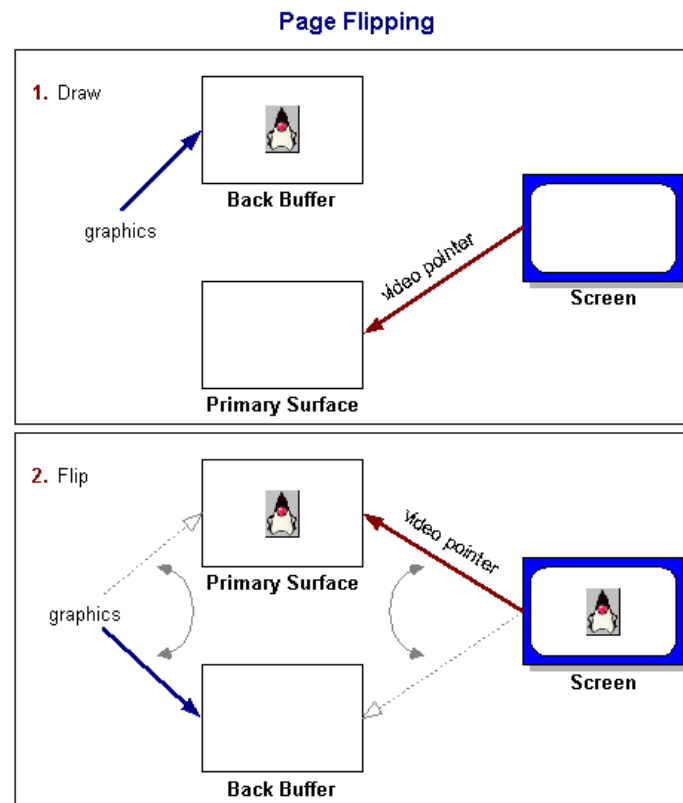


oracle.com



# Double Buffering

- In fullscreen mode, only the video pointer is flipped:



oracle.com

# Managing User Interaction

- The user may interact with the program through input devices: traditionally keyboard and mouse
- GLUT also simplifies this task by registering callbacks:

```
glutReshapeFunc( reshape );  
glutKeyboardFunc( keyboard );
```

# Managing User Interaction

- Sample keyboard callback:

```
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 27: // Escape
            exit(0);
            break;
        case 'q': // normal key press
            exit(0);
            break;
        default:
            break;
    }
}
```

# Managing User Interaction

- Sample special key-press callback:

```
void special(int key, int x, int y)
{
    switch (key)
    {
        case GLUT_KEY_LEFT:
            break;
        case GLUT_KEY_RIGHT:
            break;
        case GLUT_KEY_UP:
            break;
        case GLUT_KEY_DOWN:
            break;
        default:
            break;
    }
}
```

# Displaying/Resizing the Window

- Whenever a window is displayed or resized, certain settings (such as the viewport) may need to be updated:
- This function can also be registered by using GLUT:

```
glutReshapeFunc( reshape );
```

# Displaying/Resizing the Window

- Here, we typically reset the viewport and transformation matrices:

```
void reshape(int w, int h)
{
    w = w < 1 ? 1 : w;
    h = h < 1 ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Window width and height

Projection transform can be set by *glOrtho* or *glFrustum*. It is also possible to use *gluPerspective*

Combined modeling and viewing transform

# Rendering Each Frame

- **Each frame must be redrawn from scratch!**
- Again, we first register a callback for this task
- The registered function is automatically called by the windowing system whenever required:

```
glutDisplayFunc(display);
```

# Rendering Each Frame

- We first clear all buffers, then render our frame, and finally swap buffers (remember double buffering):

```
void display()
{
    glClearColor(0, 0, 0, 1);
    glClearDepth(1.0f);
    glClearStencil(0);
    glClear(GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT |
            GL_STENCIL_BUFFER_BIT);

    renderFrame();

    glutSwapBuffers();
}
```



# Animation


- If we have **animation**, we must make sure that the window system calls our display function continuously
- For that purpose, we register another callback:

```
glutIdleFunc(idle);
```

- In this function, we simply ask our display function to be called during GLUT's main loop:

```
void idle()  
{  
    glutPostRedisplay();  
}
```

Sets a flag so that  
our display function  
will be called



# Sending Geometry Data

- The user program must communicate the geometry information to the GPU
- A simple approach:

```
glBegin(GL_LINES);  
    glVertex3f(x0, y0, z0);  
    glVertex3f(x1, y1, z1);  
glEnd();
```

- We tell GPU that we want to draw a line from  $(x_0, y_0, z_0)$  to  $(x_1, y_1, z_1)$

# Sending Geometry Data

- Attributes besides position can be sent as well:

```
glBegin(GL_LINES);  
    glColor3f(1, 0, 0); // red  
    glVertex3f(x0, y0, z0);  
    glColor3f(0, 1, 0); // green  
    glVertex3f(x1, y1, z1);  
glEnd();
```

- We tell GPU that we want to draw a line from  $(x_0, y_0, z_0)$  to  $(x_1, y_1, z_1)$
- The endpoint colors are  $(1, 0, 0)$  and  $(0, 1, 0)$

# Sending Geometry Data

- Triangles are similar:

```
glBegin(GL_TRIANGLES);  
    glVertex3f(x0, y0, z0);  
    glVertex3f(x1, y1, z1);  
    glVertex3f(x2, y2, z2);  
glEnd();
```

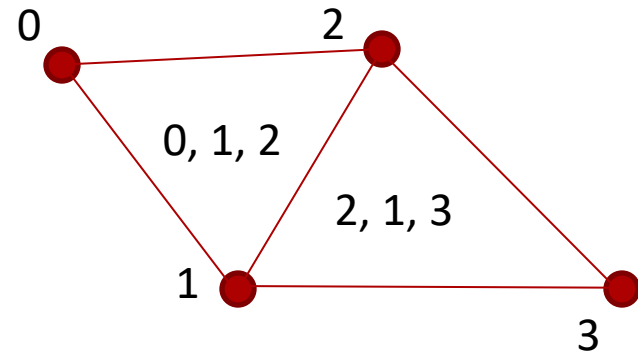
- Every group of **three vertices** define a triangle
- Drawing two triangles:

```
glBegin(GL_TRIANGLES);  
    glVertex3f(x0, y0, z0); glVertex3f(x1, y1, z1); glVertex3f(x2, y2, z2);  
    glVertex3f(x3, y3, z3); glVertex3f(x4, y4, z4); glVertex3f(x5, y5, z5);  
glEnd();
```

# Sending Geometry Data

- With this approach  $m$  triangles require  $3m$  vertex calls
- An improved method is to use **triangle strips** for meshes
- The first three vertices define the first triangle
- Every vertex afterwards defines a new triangle

```
glBegin(GL_TRIANGLE_STRIP);  
  glVertex3f(x0, y0, z0);  
  glVertex3f(x1, y1, z1);  
  glVertex3f(x2, y2, z2);  
  glVertex3f(x3, y3, z3);  
glEnd();
```

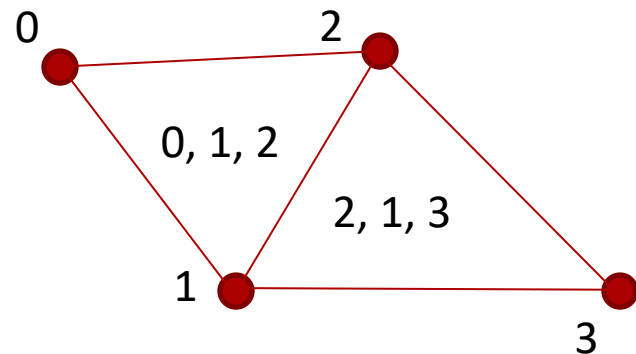


- $m$  triangles require  $m+2$  vertex calls

# Winding Order

- **Winding order** determines the facing of a triangle
- Here both triangles are facing toward the viewer:

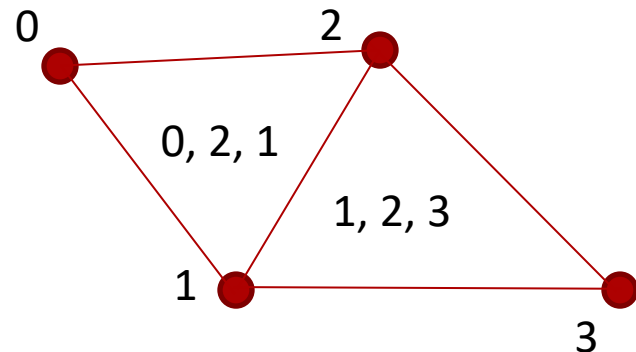
```
glBegin(GL_TRIANGLE_STRIP);  
  glVertex3f(x0, y0, z0);  
  glVertex3f(x1, y1, z1);  
  glVertex3f(x2, y2, z2);  
  glVertex3f(x3, y3, z3);  
glEnd();
```



# Winding Order

- **Winding order** determines the facing of a triangle
- Here both triangles are facing away from the viewer:

```
glBegin(GL_TRIANGLE_STRIP);  
  glVertex3f(x0, y0, z0);  
  glVertex3f(x2, y2, z2);  
  glVertex3f(x1, y1, z1);  
  glVertex3f(x3, y3, z3);  
glEnd();
```



- It is important to use a consistent winding order when drawing a mesh due to backface culling

# Graphics State

- OpenGL is a **state machine**
- Various states are preserved until we change them
- In the example below, the color of each vertex is set to (0, 1, 0), that is green:

```
glColor3f(0, 1, 0)
glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(x0, y0, z0);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y2, z2);
    glVertex3f(x3, y3, z3);
glEnd();
```



# Graphics State

- Below the first three vertices have the same color and normal
- The fourth vertex has a different color and normal:

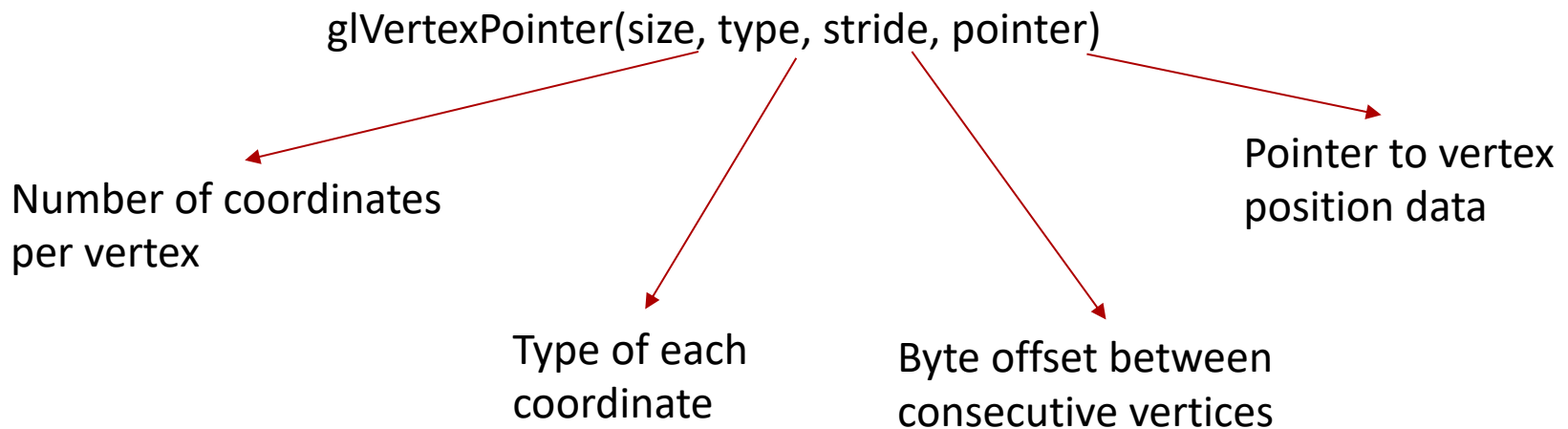
```
glColor3f(0, 1, 0)
glNormal3f(0, 0, 1)
glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(x0, y0, z0);
    glVertex3f(x2, y2, z2);
    glVertex3f(x1, y1, z1);
    glColor3f(1, 0, 0)
    glNormal3f(1, 0, 1)
    glVertex3f(x3, y3, z3);
glEnd();
```

# Sending Geometry Data

- Previous examples send data in **immediate mode**
- **Immediate mode is inefficient:** A large model would require too many glVertex calls
- Each glVertex call is executed on the CPU and the corresponding data is sent to the GPU
- A better approach would be to send all vertex data to the GPU using a single call
- We use **vertex arrays** for that purpose

# Vertex Arrays

- There are several arrays such as vertex position array, vertex color array, vertex normal array, ...
- Below is an example of vertex position array:



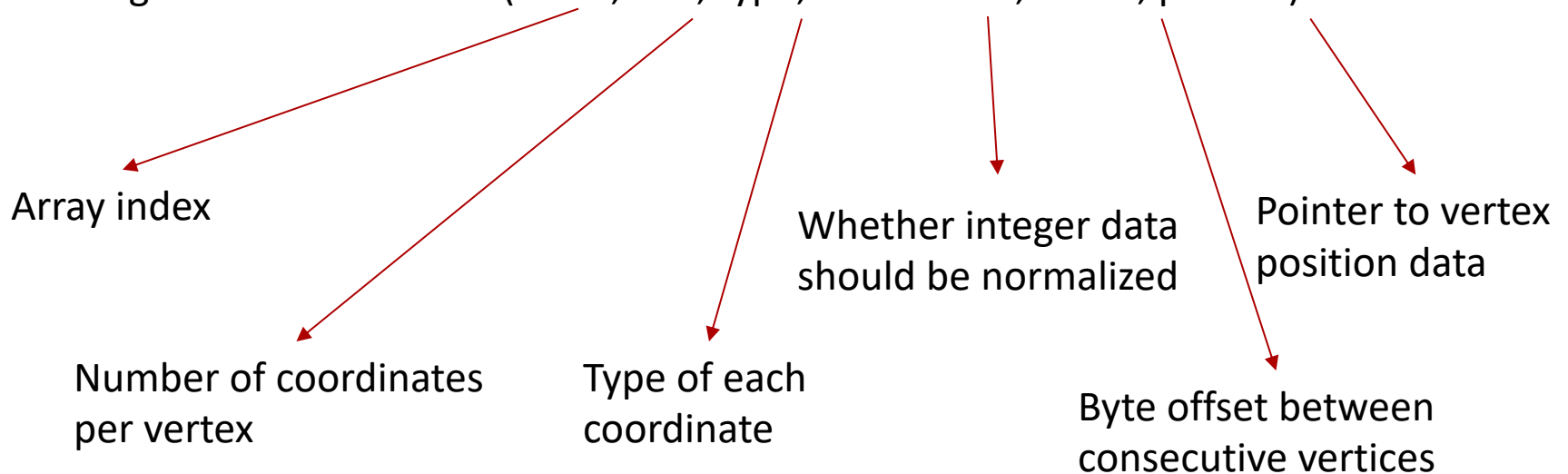
- You must enable an array before using it:

`glEnableClientState(GL_VERTEX_ARRAY)`

# Vertex Arrays

- In modern OpenGL, these **explicit** attribute names are replaced by a **generic** attribute array function:

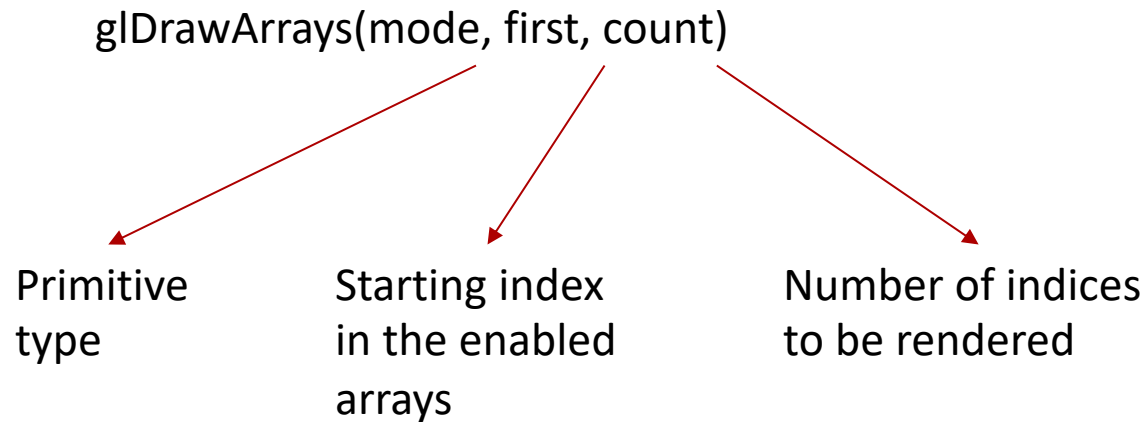
`glVertexAttribPointer(index, size, type, normalized, stride, pointer)`



- Don't forget to enable it: `glEnableVertexAttribArray(index)`

# Drawing with Vertex Arrays

- We use a single draw call to draw using vertex arrays:



# Drawing with Vertex Arrays

- **glDrawArrays** may still be inefficient as vertex attribute data must be repeated for each primitive
- **glDrawElements** is designed to solve this issue by using indices:

`glDrawElements(mode, count, type, indices)`

Primitive  
type

Number of indices  
to be rendered

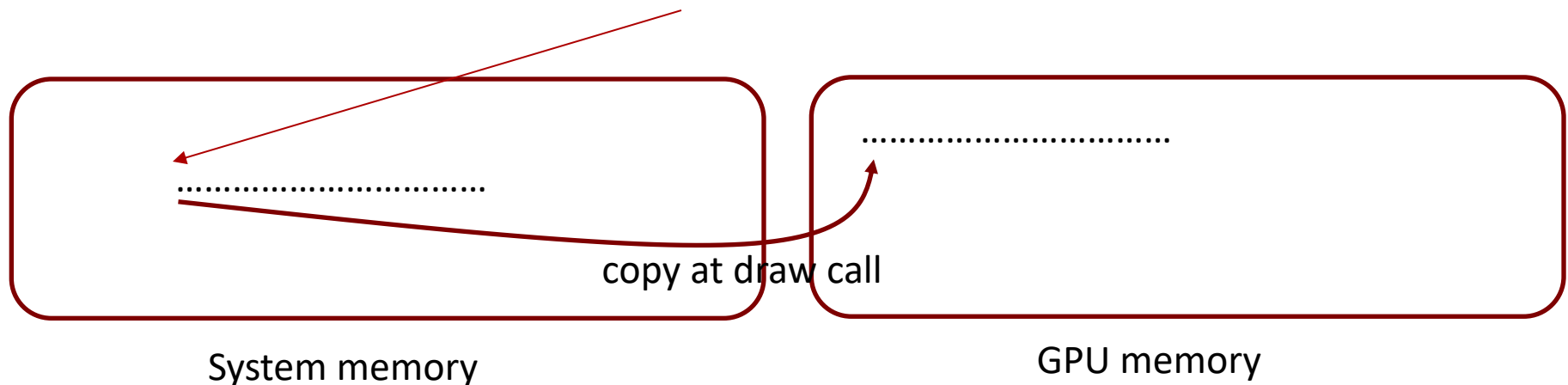
Type of  
each index

Pointer to  
indices

# Drawing with Vertex Arrays

- When using client-side vertex arrays, the vertex attribute data is copied from the system memory (user pointer) to the GPU memory at every draw call
- There is a better alternative, known as **vertex buffers**

`glVertexPointer(size, type, stride, pointer)`



# Vertex Buffer Objects

- Previous methods required the data to be copied from the system memory to GPU memory at each draw
- Vertex Buffer Objects (VBOs) are designed to allow this copy to take place only once
- The copied data is reused at each draw



# Vertex Buffer Objects

- To use VBOs, we generate two buffers:
  - Vertex attribute buffer (position, color, normal, etc.)
  - Element array buffer (indices)

```
GLuint vertexAttribBuffer, indexBuffer;
```

```
glGenBuffers(1, &vertexAttribBuffer);  
glGenBuffers(1, &indexBuffer);
```

# Vertex Buffer Objects

- Next, we bind these buffers to locations that are meaningful for the GPU:

```
glBindBuffer(GL_ARRAY_BUFFER, vertexAttribBuffer);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer)
```

# Vertex Buffer Objects

- We then ask the GPU to allocate memory for us and copy our data into this memory

Attribute data size  
in bytes

Attribute data pointer

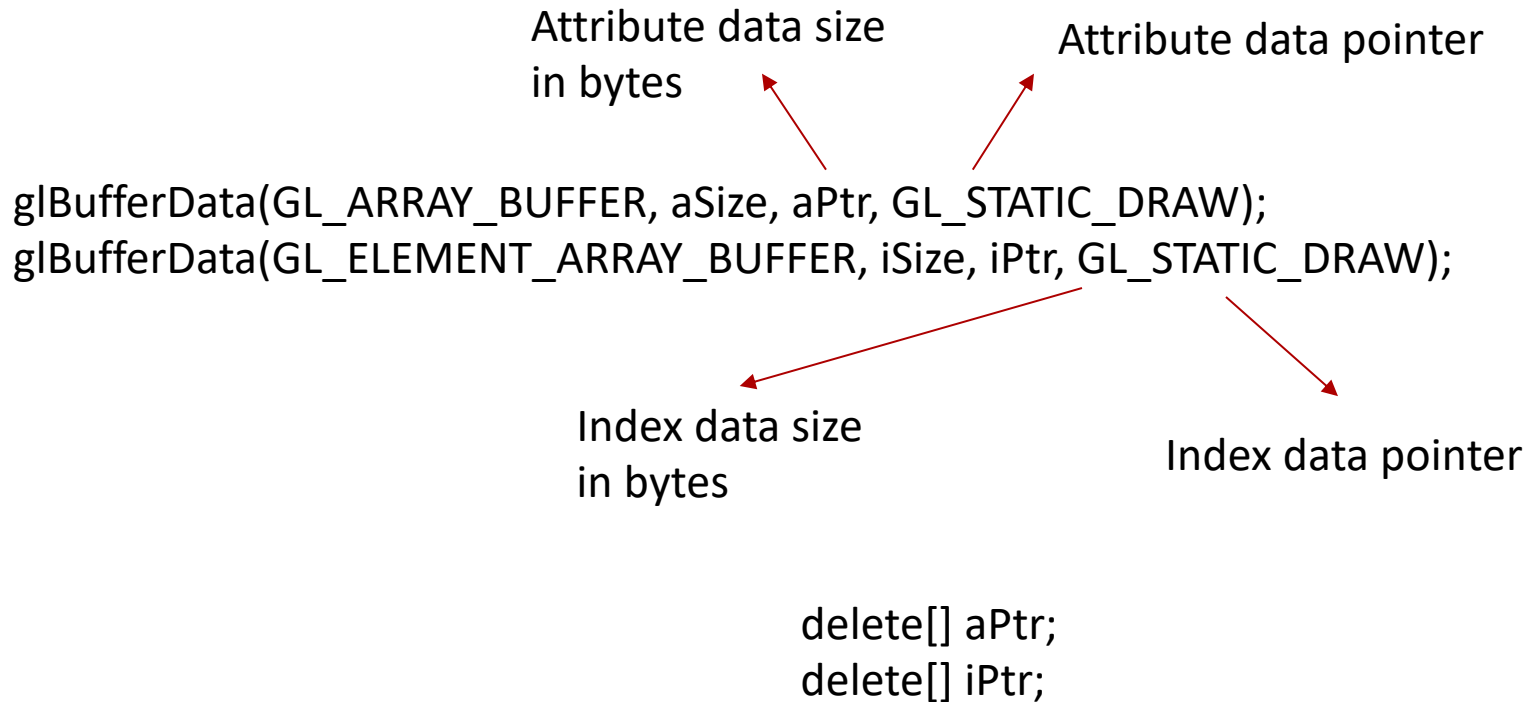
```
glBufferData(GL_ARRAY_BUFFER, aSize, aPtr, GL_STATIC_DRAW);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, iSize, iPtr, GL_STATIC_DRAW);
```

Index data size  
in bytes

Index data pointer

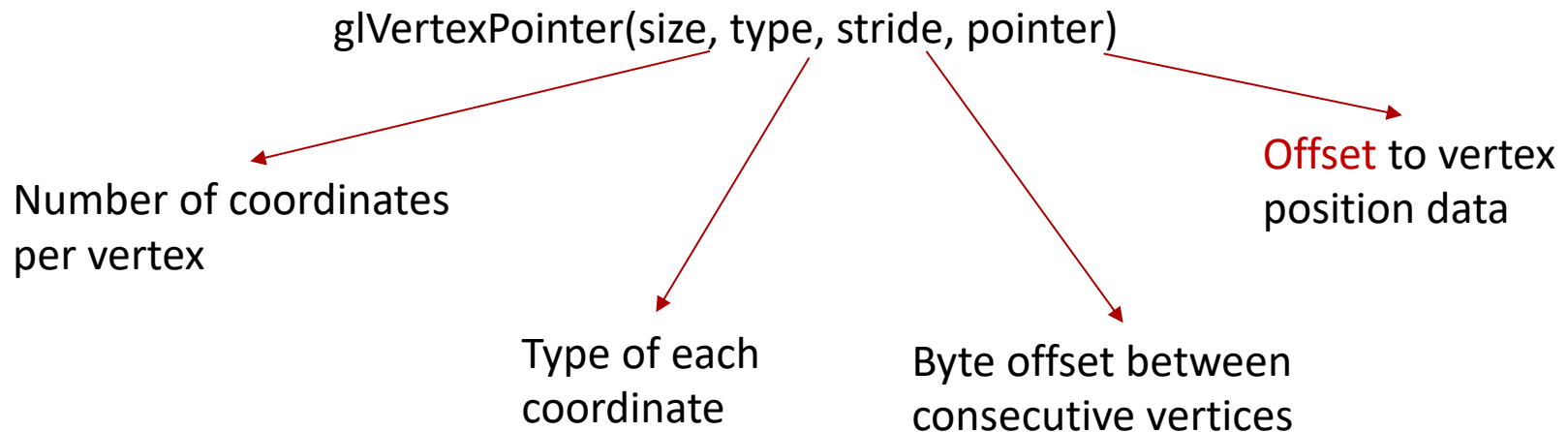
# Vertex Buffer Objects

- Once this is done, the CPU data can safely be deleted:



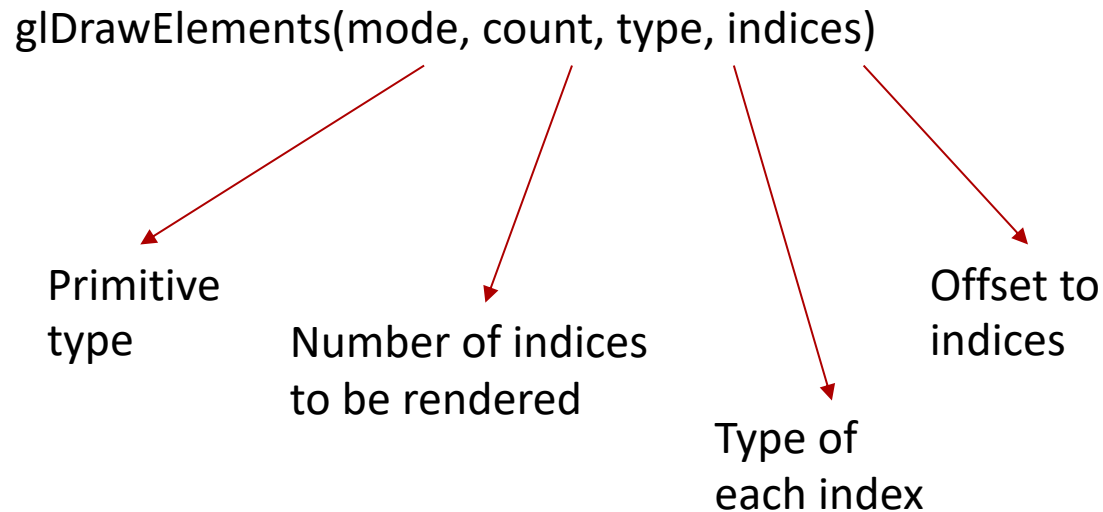
# Vertex Buffer Objects

- Before drawing, we can specify an offset into our buffers
- It is accomplished by the same function as before
- But this time, pointer indicates a byte **offset** into our buffer (similar for glColorPointer, etc.)



# Vertex Buffer Objects

- Drawing is the same as before where index pointer is now also an **offset** to the element array buffer:



# Vertex Buffer Objects

- The relevant buffers must still be enabled:

```
glEnableClientState(GL_VERTEX_ARRAY)  
glEnableClientState(GL_COLOR_ARRAY)  
...
```

- Unfortunately, this is a very bad naming as it suggests client-side data is being used
- In modern OpenGL, these are replaced with:

```
glEnableVertexAttribArray(0);  
glEnableVertexAttribArray(1);  
...
```

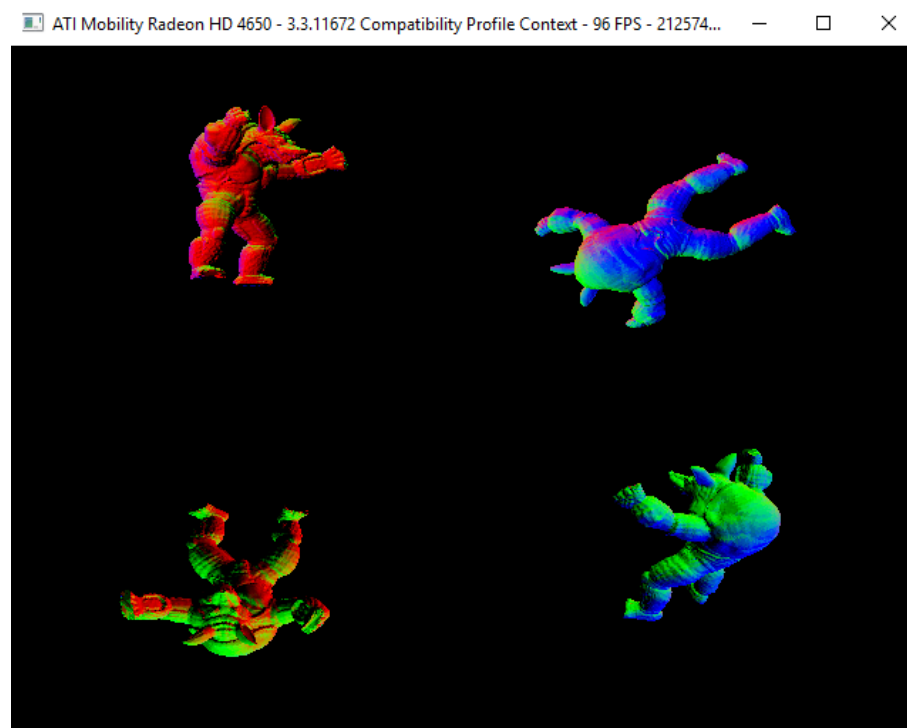
# Vertex Buffer Objects

- Note that in **glVertexPointer** and **glDrawElements** the last parameter is sometimes treated as pointer and sometimes offset
- OpenGL makes this decision as follows:
  - If a **non-zero name** is bound to **GL\_ARRAY\_BUFFER**, the last parameter glVertexPointer is treated as offset (otherwise pointer)
  - If a **non-zero name** is bound to **GL\_ELEMENT\_ARRAY\_BUFFER**, the last parameter glDrawElements is treated as offset (otherwise pointer)



# Performance Comparison

- Drawing an Armadillo model comprised of 212574 triangles at four distinct locations (resulting in a total of 850296 triangles):



# Performance Comparison

- On AMD Mobility Radeon HD4650 and at resolution 640x480:
  - Using VBOs the frame rate was about 100 FPS
  - Using client-side `glDrawElements`, the frame rate was about 20 FPS
- Therefore, almost all modern games use VBOs for drawing complex models

# Transformations in OpenGL

- In classic OpenGL, transformations are performed using three commands:

```
glTranslatef(deltaX, deltaY, deltaZ);  
glRotatef(angle, axisX, axisY, axisZ);  
glScalef(scaleX, scaleY, scaleZ);
```

- These commands effect the current matrix
- Therefore the current matrix should be set as **GL\_MODELVIEW** before calling these commands
- Note that **angle** is in degrees (not radians)!

# Transformations in OpenGL

- Transformations apply in the **reverse order**
- The command closest to the *draw* call takes effect first

```
glTranslatef(deltaX, deltaY, deltaZ);  
glRotatef(angle, axisX, axisY, axisZ);  
glScalef(scaleX, scaleY, scaleZ);
```

```
drawCube();
```

- Here, the cube is first scaled, then rotated, and finally translated

# Transformations in OpenGL

- Transformations keep effecting the current matrix
- If you want to draw an object at the same position at each frame you need to reset the matrix to identity:

```
glLoadIdentity();
```

```
glTranslatef(deltaX, deltaY, deltaZ);  
glRotatef(angle, axisX, axisY, axisZ);  
glScalef(scaleX, scaleY, scaleZ);
```

```
drawCube();
```

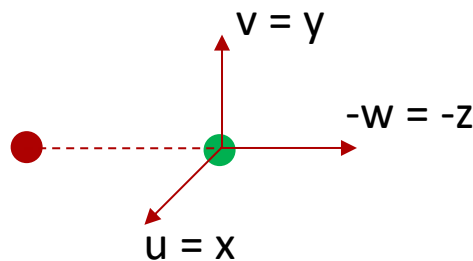
- Otherwise your object will quickly disappear!

# Transformations in OpenGL

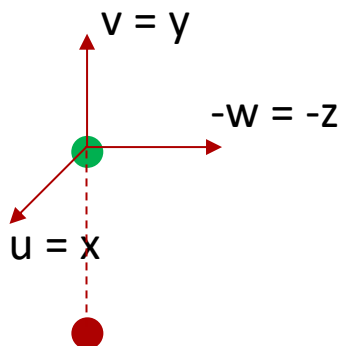
- In OpenGL, we do not specify a camera position
- It is assumed that the camera is at **(0, 0, 0)** and looking down the **negative z** axis
- You can view a modelview transformation in two ways:
  - Transform all objects drawn after the transformation by keeping the camera fixed
  - Transform the camera (i.e. coordinate system) by the opposite transformations by keeping the objects fixed
- In reality, objects are transformed but both would produce the same result

# Transformations in OpenGL

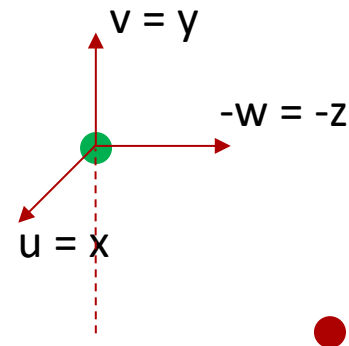
- Assume we have an object at (0, 0, 4):



Apply `glRotatef(90, 1, 0, 0)`



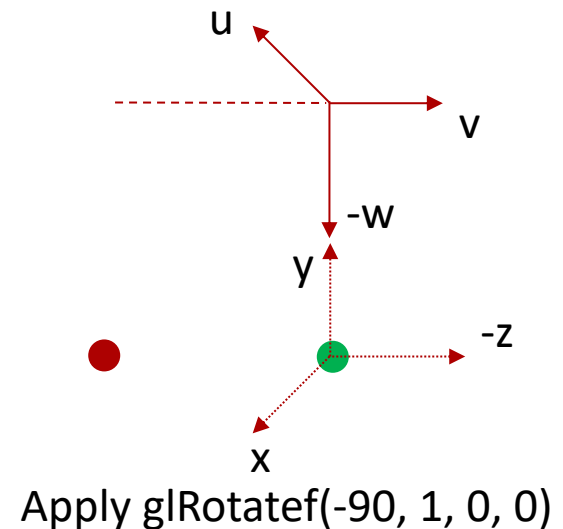
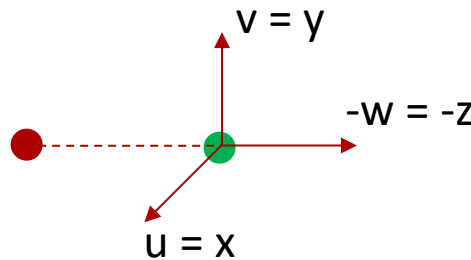
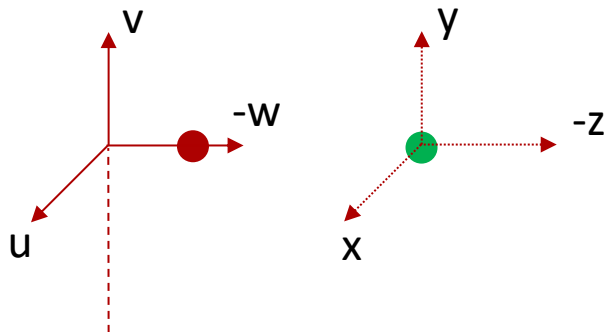
Apply `glTranslatef(0, 0, -5)`



# Transformations in OpenGL

- Now imagine applying the opposite to the camera:

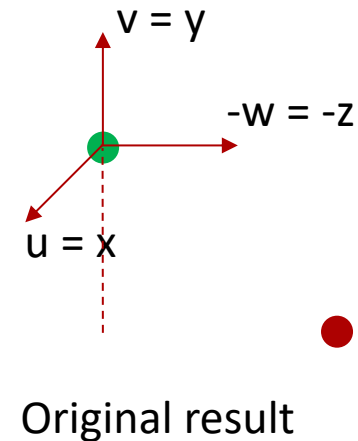
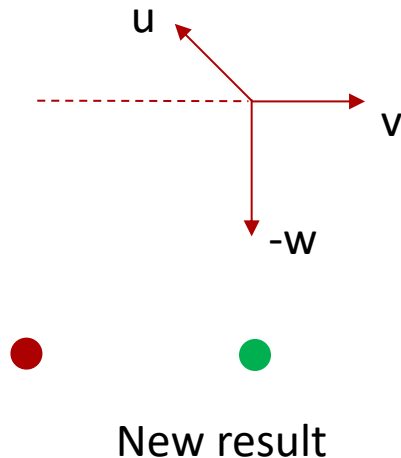
Apply `glTranslatef(0, 0, 5)`





# Transformations in OpenGL

- Now imagine applying the opposite to the camera:



The object position w.r.t. the camera  
is exactly the same in these two cases