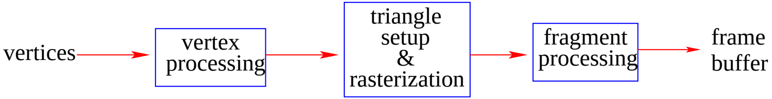
OpenGL 4
Introduction to Computer Graphics

Andrzej Szymczak

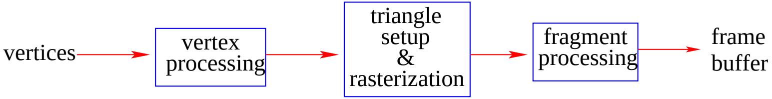
October 12, 2012

Motivation



- The old way requires sending a lot of commands from the CPU to the GPU
  - CPU could waste time converting data/commands into the right format
  - More communication than necessary
  - Let GPU operate on 'objects' contained in its own memory
  - Better send large blocks of data using a small number of OpenGL calls (e.g. all vertices at once vs one at a time as in OpenGL 1 `glVertex` function)
- Since programmable shaders are available:
  - Why hard code vertex attributes any more? Give the programmer control over what they are
  - *Require* the programmer provide shaders; after all, the traditional functionality can be replicated this way

Specifying vertices: the new way

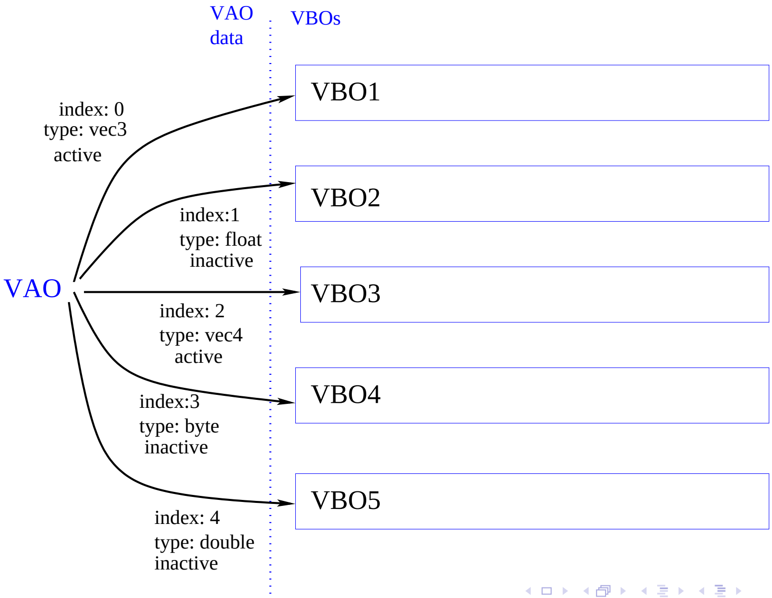


- In many cases, the models you render don't change for a number of frames
- Keep the vertex data in the GPU's memory to save time (less CPU-GPU communication)
- Generic attributes
  - Let the programmer specify any number of attributes of different types for each vertex
  - Each attribute will have an *index*
  - Programmer-provided vertex shader will recognize attributes by index and interpret them properly

Specifying vertices: VAOs and VBOs

- Vertex Buffer Objects
  - Arrays of vertex properties (coordinates, normals, colors or anything else) stored in the GPU memory
  - Identified by a VBO handle, an unsigned integer (GLuint type)
- Vertex Array Objects
  - Think of them as bundles of VBOs, with instructions on how to build vertices from VBOs' contents
  - Identified by a VAO handle, a GLuint type value (note: this generally the case for all OpenGL 'objects')
  - Each VBO in the bundle holds a different vertex attribute
  - Attributes have *indices*, integer identifiers
  - Different attributes may have different types or number of components (e.g. can be 4D vectors, 2D vectors, floats, integers)
  - Generally, when creating a 'vertex', the GPU will assemble its attributes by looking them up from all active VBOs for the current active VAO

VAOs and VBOs



Creating a useful VAO

- Generate a handle for the VAO
  - `GLuint vao;`
  - `glGenVertexArrays(1,&vao);` // first arg is # handles to generate

Creating a useful VAO

- Generate a handle for the VAO
  - `GLuint vao;`
  - `glGenVertexArrays(1,&vao);` // first arg is # handles to generate
- Bind the VAO; basically, this means that you want to work on/with your VAO now
  - `glBindVertexArray(vao);`

Creating a useful VAO

- Generate a handle for the VAO
  - `GLuint vao;`
  - `glGenVertexArrays(1,&vao);` // first arg is # handles to generate
- Bind the VAO; basically, this means that you want to work on/with your VAO now
  - `glBindVertexArray(vao);`
- Now, we need to attach some VBOs to our VAO
  - Here we'll need one for coordinates and one for normals
  - Start off by creating handles for them
  - `GLuint vbo[2];`
  - `glGenBuffers(2,vbo);` //note similarity to `glGenVertexArrays`

Creating a useful VAO

- Generate a handle for the VAO
  - `GLuint vao;`
  - `glGenVertexArrays(1,&vao);` // first arg is # handles to generate
- Bind the VAO; basically, this means that you want to work on/with your VAO now
  - `glBindVertexArray(vao);`
- Now, we need to attach some VBOs to our VAO
  - Here we'll need one for coordinates and one for normals
  - Start off by creating handles for them
  - `GLuint vbo[2];`
  - `glGenBuffers(2,vbo);` //note similarity to `glGenVertexArrays`
- Bind the first VBO
  - `glBindBuffer(GL_ARRAY_BUFFER,vbo[0]);`

Creating a useful VAO

- Generate a handle for the VAO
  - `GLuint vao;`
  - `glGenVertexArrays(1,&vao);` // first arg is # handles to generate
- Bind the VAO; basically, this means that you want to work on/with your VAO now
  - `glBindVertexArray(vao);`
- Now, we need to attach some VBOs to our VAO
  - Here we'll need one for coordinates and one for normals
  - Start off by creating handles for them
  - `GLuint vbo[2];`
  - `glGenBuffers(2,vbo);` //note similarity to `glGenVertexArrays`
- Bind the first VBO
  - `glBindBuffer(GL_ARRAY_BUFFER,vbo[0]);`
- Send data from an array in the main memory to the GPU memory
  - `glBufferData(GL_ARRAY_BUFFER, 12*3*3*sizeof(GLfloat),coordarray,GL_STATIC_DRAW);`

Creating a useful VAO, cont.

- The current VBO (vbo[0]) stores attribute
  - with index 0
  - with 3 coordinates per vertex
  - of GLfloat type
  - there is no need to normalize it
  - Here is how to put this info into the VAO:
    `glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);`

Creating a useful VAO, cont.

- The current VBO (vbo[0]) stores attribute
  - with index 0
  - with 3 coordinates per vertex
  - of GLfloat type
  - there is no need to normalize it
  - Here is how to put this info into the VAO:
    `glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);`
- Bind the second VBO and send data (normals) to it; this is attribute of index 1
  - `glBindBuffer(GL_ARRAY_BUFFER,vbo[1]);`
  - `glBufferData(GL_ARRAY_BUFFER, 12*3*3*sizeof(GLfloat),normalarray,GL_STATIC_DRAW);`
  - `glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,0,0);`

Creating a useful VAO, cont.

- The current VBO (vbo[0]) stores attribute
  - with index 0
  - with 3 coordinates per vertex
  - of GLfloat type
  - there is no need to normalize it
  - Here is how to put this info into the VAO:
    `glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);`
- Bind the second VBO and send data (normals) to it; this is attribute of index 1
  - `glBindBuffer(GL_ARRAY_BUFFER,vbo[1]);`
  - `glBufferData(GL_ARRAY_BUFFER, 12*3*3*sizeof(GLfloat),normalarray,GL_STATIC_DRAW);`
  - `glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,0,0);`
- Enable both attributes
  - `glEnableVertexAttribArray(0);`
  - `glEnableVertexAttribArray(1);`

Creating a useful VAO, cont.

- Finally, unbind the VBO and VAO – we are done working with them
  - `glBindBuffer(GL_ARRAY_BUFFER,0);`
  - `glBindVertexArray(0);`

Using our VAO

- Reminder: the goal is to send these vertices into the pipeline
- First, bind the VAO
  - `glBindVertexArray(vao);`
- Send the vertices
  - `glDrawArrays(GL_TRIANGLES,0,12*3);`
  - First argument: how to interpret the vertex stream (relevant to primitive setup)
  - Next two: range of indices to use (note that the same index i is used to pick attributes of the vertex from all VBOs)
- Unbind the VAO when done
  - `glBindVertexArray(0);`

Generalities

- OpenGL is full of different kinds of objects
- They store data used for rendering and/or some state, basically information on how to interpret the data
- Working with objects of any type generally requires similar steps
  - Bind
  - Work
  - Unbind
- The sample code does not free memory used by objects!
- OK since it only creates a small number of objects at startup
  - In your project, do the same!
  - Don't create unnecessary objects
  - Don't keep *recreating* something that already exists!
  - ... especially for each frame (you'll have a memory leak)
- In more complex code, you need to delete objects (such as programs, shaders, VBOs, VAOs etc)
  - See `glDelete*` functions in OpenGL manual pages

Vertex program

- Capture input vertex attributes; location qualifier provides the index information
  - `layout(location=0) in vec3 coord;`
  - `layout(location=1) in vec3 normal;`
- We'll send just one output attribute with the processed vertex
  - `flat` qualifier requests flat 'interpolation'; alternatives: `noperspective` or `smooth`
  - `flat out float NdotL;`
- Declare uniform variables – they can be set from the C code using `glUniform*`
  - `uniform mat4 ModelViewMatrix;`
  - `uniform mat4 ProjectionMatrix;`
  - `uniform mat3 NormalMatrix;`
  - `uniform vec3 LightLocation;`
  - Note uniform variables can't be changed from the shader

Vertex program

- Main function
  - Goals
    - Compute values of output variables (here: `NdotL`)
    - Assign coordinates of the projected vertex (in the homogenous coordinates – typically you don't want to divide by the homogenous coordinate here) to the built-in output variable `vec4 gl_Position`
    - Note that `gl_Position` is used for clipping and rasterization
  - Code:
    - `void main() {`
    - `  vec4 WorldCoord = ModelViewMatrix * vec4(coord,1.0);`
    - `  vec3 L = normalize(LightLocation - WorldCoord.xyz);`
    - `  vec3 WorldNormal = NormalMatrix*normal;`
    - `  vec3 N = normalize(WorldNormal);`
    - `  NdotL = dot(N,L);`
    - `  gl_Position = ProjectionMatrix*WorldCoord;`
    - `}`

Fragment Program

- Capture *interpolated* output of the vertex shader
  - This is how vertex and fragment shaders communicate!
  - Names and qualifiers need to match
  - `flat in float NdotL;`
  - recall that in the vertex shader we had `flat out float NdotL;`

Fragment Program

- Capture *interpolated* output of the vertex shader
  - This is how vertex and fragment shaders communicate!
  - Names and qualifiers need to match
  - `flat in float NdotL;`
  - recall that in the vertex shader we had `flat out float NdotL;`
- Declare all uniform variables (they can be set from CPU)
  - `uniform float LightIntensity;`
  - `uniform float AmbientIntensity;`
  - `uniform vec3 DiffuseAndAmbientCoefficient;`

Fragment Program

- Capture *interpolated* output of the vertex shader
  - This is how vertex and fragment shaders communicate!
  - Names and qualifiers need to match
  - `flat in float NdotL;`
  - recall that in the vertex shader we had `flat out float NdotL;`
- Declare all uniform variables (they can be set from CPU)
  - `uniform float LightIntensity;`
  - `uniform float AmbientIntensity;`
  - `uniform vec3 DiffuseAndAmbientCoefficient;`
- Declare the output variables; if there is exactly one of type vec4, the components will be treated as the RGBA values for the fragment
  - `out vec4 fragcolor;`

Fragment Program

- The main function is very simple:
  - `void main() {`
  - `fragcolor = vec4( (LightIntensity* (NdotL > 0.0 ?  NdotL : 0.0) + AmbientIntensity) * DiffuseAndAmbientCoefficient, 1);`
  - `}`

Compiling and linking vertex and fragment programs

- Create handles
  - `GLuint vid = glCreateShader(GL_VERTEX_SHADER);`
  - `GLuint fid = glCreateShader(GL_FRAGMENT_SHADER);`
- Specify source
  - `glShaderSource(vid,1,&VS,NULL);`
  - `glShaderSource(fid,1,&FS,NULL);`
  - This works if VS and FS are C-style null-terminated strings (char* type)
- Compile shaders
  - `glCompileShader(vid);`
  - `glCompileShader(fid);`
- Create program handle (it will include complete set of shaders)
  - `GLuint ProgramHandle = glCreateProgram();`
- Attach our vertex and fragment shaders to the program
  - `glAttachShader(ProgramHandle,vid);`
  - `glAttachShader(ProgramHandle,fid);`

Compiling and linking vertex and fragment programs

- Link
  - `glLinkProgram(ProgramHandle);`
  - This is where vertex shader outputs are matched with fragment shader outputs and any shared uniform variables are merged

Using the program

- `glUseProgram(ProgramHandle);`
- Now, the program will be applied to any vertex stream
- In general, you may have several programs, e.g. in your project you may want to write one for flat shading and one for Gouraud shading; you can use `glUseProgram` to switch between different programs – just pass the handle of the program you want to use as the argument

Uniform variables

- Each uniform variable has a *location* in the program
- Use `glGetUniformLocation` to get locations of uniforms
  - Example: `GLint NormalMatrixLoc = glGetUniformLocation(ProgramHandle,"NormalMatrix");`
  - Return value is -1 means variable was not found
    - Could mean a typo
    - ... or that the compiler figured out that the uniform variable can be optimized out

Uniform variables

- Each uniform variable has a *location* in the program
- Use `glGetUniformLocation` to get locations of uniforms
  - Example: `GLint NormalMatrixLoc = glGetUniformLocation(ProgramHandle,"NormalMatrix");`
  - Return value is -1 means variable was not found
    - Could mean a typo
    - ... or that the compiler figured out that the uniform variable can be optimized out
- Set values of uniform variables using `glUniform*`
  - The program has to be active (call `glUseProgram` before!)
  - `glUniform1f(LightIntensityLoc,0.9);`
  - `glUniform3f(DiffuseAndAmbientCoefficientLoc, 0.0,1.0,0.0);` (this one is of vec3 type)
  - Alternative: `glUniform3fv` – use pointer to 3D array of floats as second argument
  - For matrix uniforms, pass pointer to an array of entries (here, array of size 16 since the matrix is 4×4) `glUniformMatrix4fv(ModelViewMatrixLoc,1,GL_FALSE,ptr);`

GLM library

- A simple header-only library that provides useful math functions
  - Vector operations
  - Matrix operations
  - Replicates functionality of most useful obsolete OpenGL functions such as transformation calls
  - Also, replicates most useful deprecated GL utilities (GLU) functions

Using GLM to set matrices

- Setting the projection matrix

```
glm::mat4 PMat =
.     glm::perspective(8.0f,1.0f,15.0f,25.0f);
glUniformMatrix4fv(ProjectionMatrixLoc,1,
.     GL_FALSE,&PMat[0][0]);
```

- Setting the modelview matrix

```
glm::mat4 MVMat = glm::translate(glm::mat4(1.0),
.     glm::vec3(0.0,0.0,-20.0)) *
.     glm::rotate(glm::mat4(1.0),angle1,
.         glm::vec3(1.0,2.0,3.0)) *
.     glm::rotate(glm::mat4(1.0),angle2,
.         glm::vec3(-2.0,-1.0,0.0));
glUniformMatrix4fv(ModelViewMatrixLoc,1,GL_FALSE,
.     &MVMat[0][0]);
```

Using GLM to set matrices

- Setting the normal matrix
  - In this case normals are changed only by the rotational component – thus, we can use the 3×3 linear part of the modelview matrix as the normal matrix
  - In general, you would take inverse and transpose of NMat used below before sending it to the uniform variable

```
glm::mat3 NMat(MVMat);
glUniformMatrix3fv(NormalMatrixLoc,1,
.    GL_FALSE,&NMat[0][0]);
```