Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

Vertex Processing and Drawing Commands

Instructor: Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS4595

Codeblock Conventions

Yellow Codeblock => Application Program (CPU)

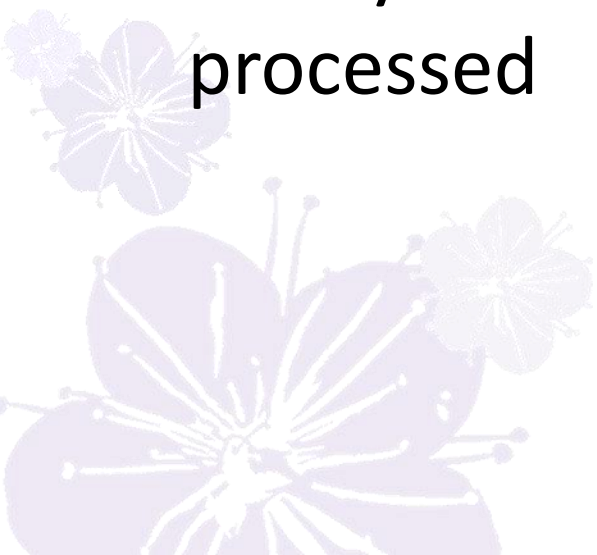
- Create OpenGL Context
- Create and Maintain OpenGL Objects
- Generate Works for the GPU to Consume

Blue Codeblock => Shader Program (GPU)

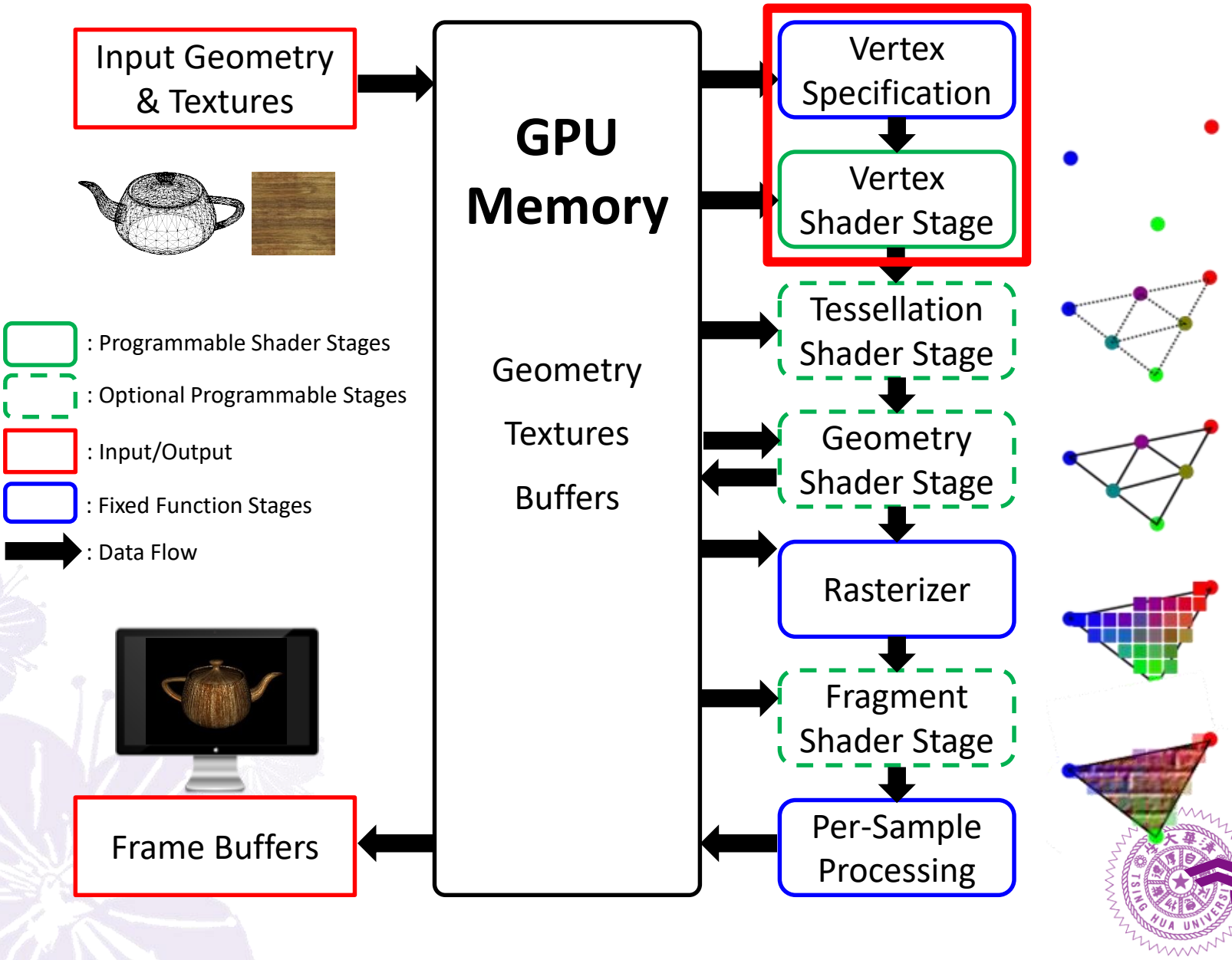
- Shader for a Certain Programmable Stage
- Process Geometry or Fragment in Parallel
- Starts with #version 410 core Declaration

What You'll Learn in This Lecture

- How to get data from your application into the front of the graphics pipeline
- What the various OpenGL drawing commands are and what their parameters do
- How your transformed geometry is post-processed



The OpenGL Rendering Pipeline

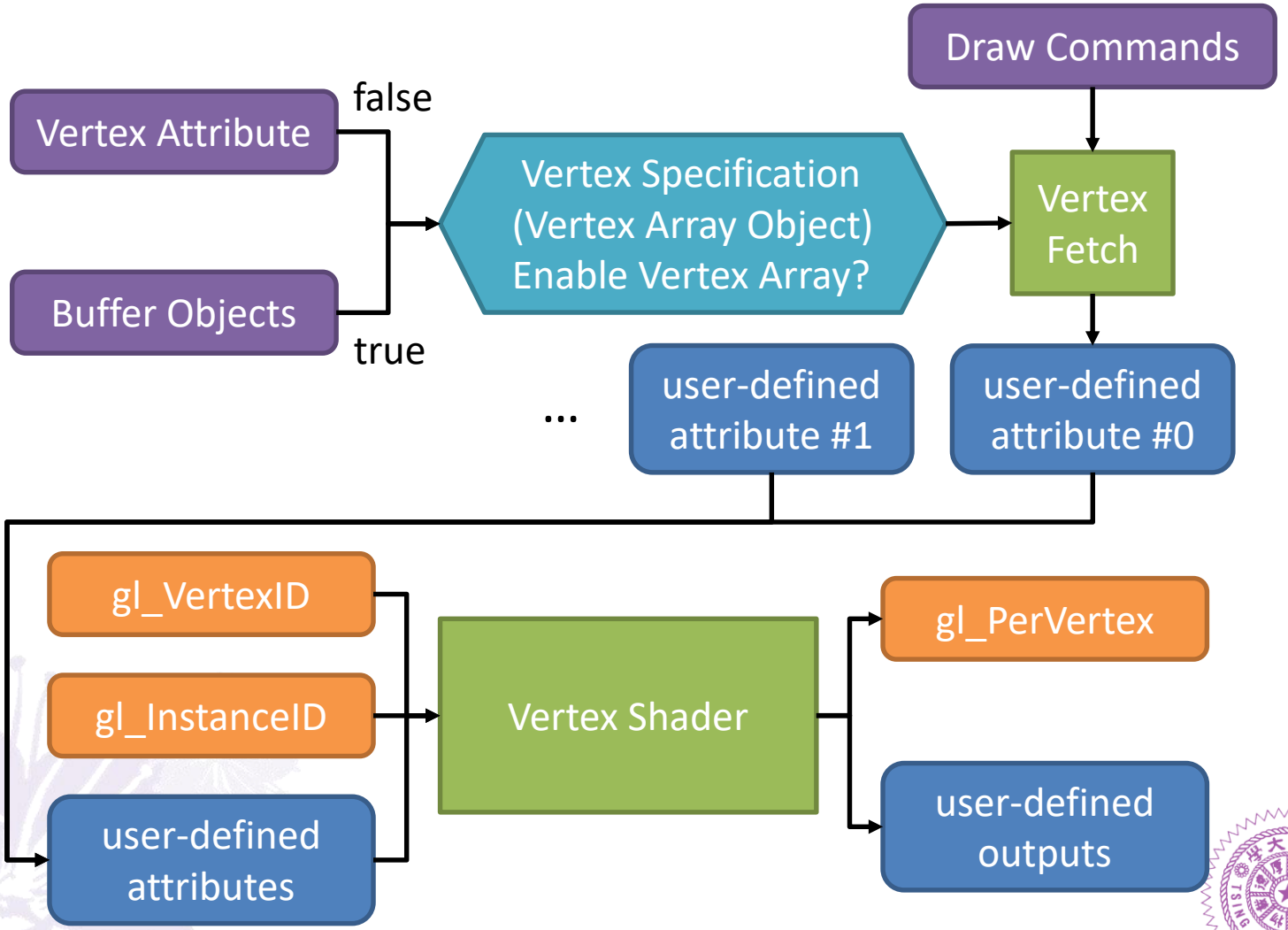


VERTEX PROCESSING

Vertex Processing

- The first programmable stage, ***vertex shader***
- Vertex shader inputs are provided by binding ***vertex array objects*** and ***issuing drawing command*** calls
- Before a vertex shader runs, OpenGL fetches its inputs in the ***vertex fetch stage***
- Vertex shaders set the ***position of the vertex*** that will be fed to the next stage, and write to some other ***user-defined and built-in outputs***

Vertex Shader Overview



Vertex Shader Inputs

- Pre-defined
  - **gl_VertexID**
  - **gl_InstanceID:** range = [**0**, instancecount)
- User-defined vertex attributes
  - for example: **in vec3 vertexPosition;**
  - another example: **in vec2 textureCoordinate;**
- The value of user-defined vertex attributes are determined in the ***vertex fetch stage***

gl_VertexID

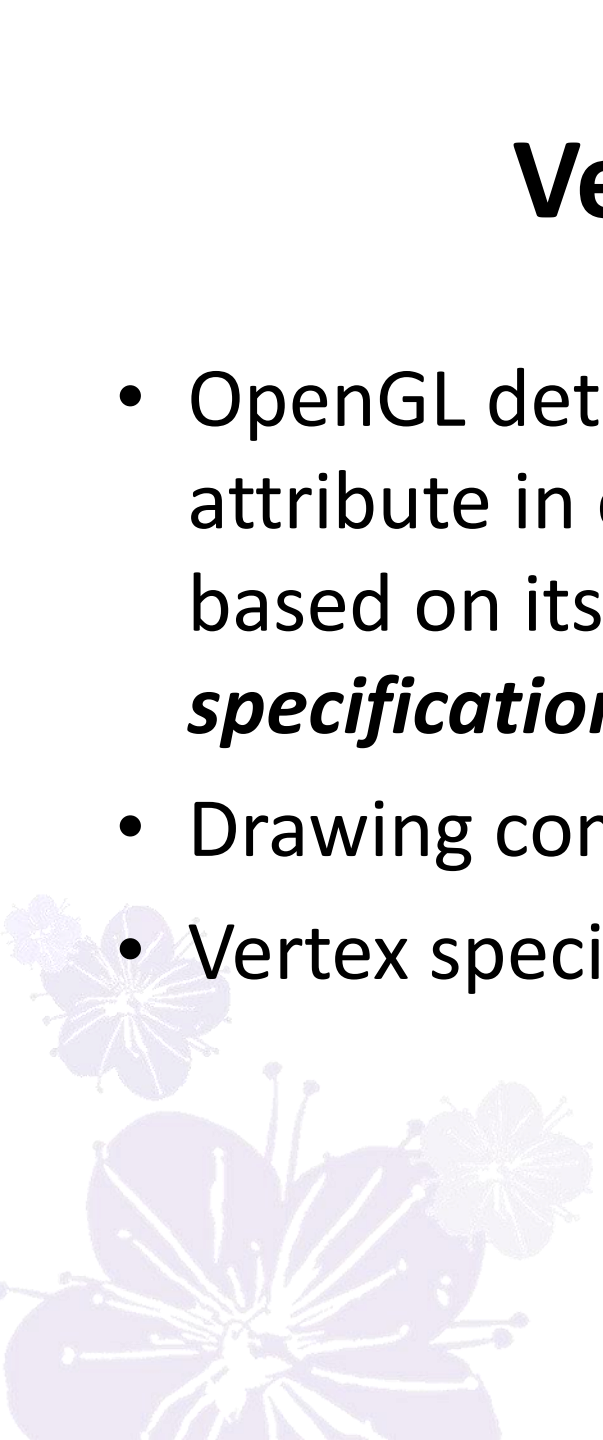
- Contains the index of the current vertex
- **Declaration**:
  - in int gl_VertexID;
- **Description**: gl_VertexID is a built-in input variable that holds an integer index for the vertex. The index is implicitly generated by glDrawArrays and other commands

gl_InstanceID

- Contains the index of the current primitive in an instanced draw command
- **Declaration**:
  - in int gl_InstanceID;
- **Description**: gl_InstanceID holds the integer index of the current primitive in an instanced draw command. Its value always starts at 0, even when using base instance calls. When not using instanced rendering, this value will be 0

Vertex Fetch Stage

- OpenGL determines the value of each vertex attribute in one vertex shader invocation based on its ***drawing command*** and ***vertex specification***
- Drawing command: ***how*** to fetch
- Vertex specification: ***where*** to fetch



Drawing Commands

- Drawing commands start ***vertex rendering***: the process of taking vertex data specified in ***vertex array object*** and rendering one or more ***primitives*** with this vertex data
- Drawing commands decide ***how*** vertex attribute data are fetched
- A bit complicated; Will be introduced later

Vertex Specification

- Vertex specification decides ***where*** vertex attribute data are fetched
- A ***bound*** and ***valid vertex array object*** must be present to store vertex specification settings

VERTEX SPECIFICATION

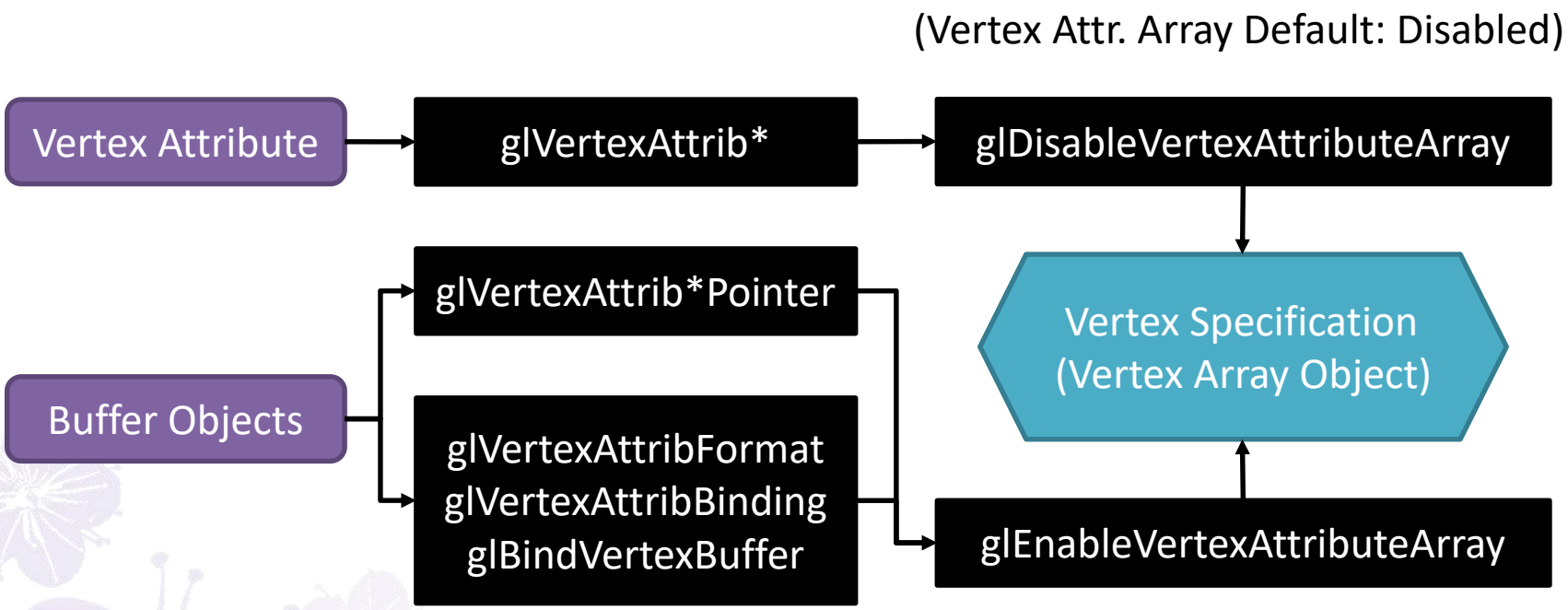
Vertex Specification

- 3 types of vertex specification APIs

1. **glVertexAttrib***
2. **glVertexAttrib*Pointer**
3. Separate attribute format binding (GL 4.3+)
   - **glVertexAttribFormat**
   - **glVertexAttribBinding**
   - **glBindVertexBuffer**

- You cannot mix type 3 with type 1 or 2!

Vertex Specification



Vertex Specification (1)

- Useful for fixing some inputs without modifying the shader

```
void render()
{
    glBindVertexArray(vao);

    glEnableVertexAttribArray(0);

    glDisableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexAttribPointer(0, 3,
GL_FLOAT, GL_FALSE, 0, 0);

    glVertexAttrib3fv(1, &color[0]);

    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 color;

layout (location = 0) uniform mat4 mvp;

out VertexData
{
    vec3 color;
} vertexData;

void main(void)
{
    vertexData.color = color;
    gl_Position =
        mvp * vec4(vertex, 1.0);
}
```

glVertexAttrib*

```
void glVertexAttrib{1234}{fds}(GLuint index, TYPE values);
void glVertexAttrib{1234}{fds}v(GLuint index,const TYPE *values);
void glVertexAttrib4{bsifd ub us ui}v(GLuint index,const TYPE
*values);
```

- **index**: Specifies the index of the generic vertex attribute to be modified.
- **values**: For the packed commands, specifies the new packed value to be used for the specified vertex attribute.

Vertex Specification (2)

- Useful for common applications

```
void render()
{
    glBindVertexArray(vao);

    glEnableVertexAttribArray(0);

    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, vbo1);
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 0, 0);

    glBindBuffer(GL_ARRAY_BUFFER, vbo2);
    glVertexAttribPointer(1, 3, GL_FLOAT,
GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 color;

layout (location = 0) uniform mat4 mvp;

out VertexData
{
    vec3 color;
} vertexData;

void main(void)
{
    vertexData.color = color;
    gl_Position =
        mvp * vec4(vertex, 1.0);
}
```

glEnableVertexAttribArray

```
void glEnableVertexAttribArray(GLuint index);
```

- **index**: Specifies the index of the generic vertex attribute to be enabled or disabled
- **Description: glEnableVertexAttribArray** enables the generic vertex attribute array specified by **index**. **glDisableVertexAttribArray** disables the generic vertex attribute array specified by **index**

glVertexAttribPointer

*void glVertexAttribPointer (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer);*

- **Description**: specify the input format of ***float*** type vertex attribute at location **index**. The attribute has **size** components and the input format is **type**. If **type** is an integer type and **normalized** is **GL_TRUE**, the value is normalized into [-1, 1] for signed and [0, 1] for unsigned. The **pointer** specifies an offset of the first component of the first generic vertex attribute. The initial value is 0.

glVertexAttrib*Pointer

*void glVertexAttribIPointer (GLuint index, GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);*

- **Description:** specify the input format of an ***integer*** type vertex attribute at location **index**

*void glVertexAttribLPointer (GLuint index, GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);*

- **Description:** specify the input format of a ***double*** type vertex attribute at location **index**

Vertex Specification (3)

- Useful for advanced binding manipulations

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    glVertexAttribFormat(0, 3, GL_FLOAT, GL_FALSE, 0);
    glVertexAttribBinding(0, 0);
    glBindVertexBuffer(0, vbo, 0, 24);

    glVertexAttribFormat(1, 3, GL_FLOAT, GL_FALSE, 12);
    glVertexAttribBinding(1, 1);
    glBindVertexBuffer(1, vbo, 0, 24);

    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Offset	Data
0	(1, 2, 3)
12	(0.5, 0.5, 0.5)
24	(2, 5, 8)

```
#version 410 core

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 color;

layout (location = 0) uniform mat4 mvp;

out VertexData
{
    vec3 color;
} vertexData;

void main(void)
{
    vertexData.color = color;
    gl_Position =
        mvp * vec4(vertex, 1.0);
}
```

glVertexAttribFormat

void glVertexAttribFormat(GLuint attribindex, GLint size, GLenum type, GLboolean normalized, Gluint relativeoffset);

- **Description:** specify the input format of a ***float*** type vertex attribute at location **attribindex**. The attribute has **size** components and the input format is **type**. If **type** is an integer type and **normalized** is **GL_TRUE**, the value is normalized into [-1, 1] for signed and [0, 1] for unsigned. The attribute is **relativeoffset** bytes from the beginning of each vertex defined in **glBindVertexBuffer**

glVertexAttrib*Format

void glVertexAttribIFormat(GLuint attribindex, GLint size, GLenum type, Gluint relativeoffset);

- **Description:** specify the input format of an ***integer*** type vertex attribute at location **attribindex**

void glVertexAttribLFormat(GLuint attribindex, GLint size, GLenum type, Gluint relativeoffset);

- **Description:** specify the input format of a ***double*** type vertex attribute at location **attribindex**

glVertexAttribBinding

void glVertexAttribBinding(GLuint attribindex, GLuint bindingindex);

- **Description:** bind a vertex attribute at location **attribindex** to a binding point **bindingindex**

glBindVertexBuffer

void glBindVertexBuffer(GLuint bindingindex, GLuint buffer, GLintptr offset, GLintptr stride);

- **Description:** bind a vertex buffer **buffer** to the binding point **bindingindex**. The data starts at **offset** bytes and the size of each vertex is **stride** bytes. If **stride** is zero, the data is assumed to be tightly-packed

Vertex Shader Outputs

- Output variables from the vertex shader are passed to the ***next stage*** of the pipeline. As some of the stages are optional, the outputs are passed to the next one that is present, in this order:

1. ***Tessellation Shaders***. If no ***Tessellation Control Shader*** is present, the ***Tessellation Evaluation Shader*** will get them.
2. ***Geometry Shader***
3. ***Vertex Post-Processing***

gl_PerVertex

- Pre-defined outputs
- ***gl_PerVertex*** defines an interface block for outputs. The block is defined without an instance name, so that prefixing the names is not required

```
out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

gl_PerVertex

- ***gl_Position***
  - The ***clip-space*** output position of the current vertex in ***homogeneous coordinate***
  - A point in object-space is transformed by a ***model-view-project matrix*** to clip-space

gl_PerVertex

- ***gl_PointSize***
  - The pixel width/height of the point being rasterized. It only has a meaning when rendering point primitives. It will be clamped to the **GL_POINT_SIZE_RANGE**
  - If **GL_PROGRAM_POINT_SIZE** is enabled, ***gl_PointSize*** is used to determine the size of rasterized points, otherwise it is ignored by the rasterization stage

gl_PerVertex

- *gl_PointSize*

```
void render()
{
    GLint sizes[2];
    glGetIntegerv(GL_POINT_SIZE_RANGE, sizes);
    printf("min point size = %d\n", sizes[0]);
    printf("max point size = %d\n", sizes[1]);

    glEnable(GL_PROGRAM_POINT_SIZE);

    glDrawArrays(GL_POINTS, 0, 3);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;

layout (location = 0) uniform mat4 mvp;

void main(void)
{
    gl_PointSize = 32;
    gl_Position =
        mvp * vec4(vertex, 1.0);
}
```

gl_PerVertex

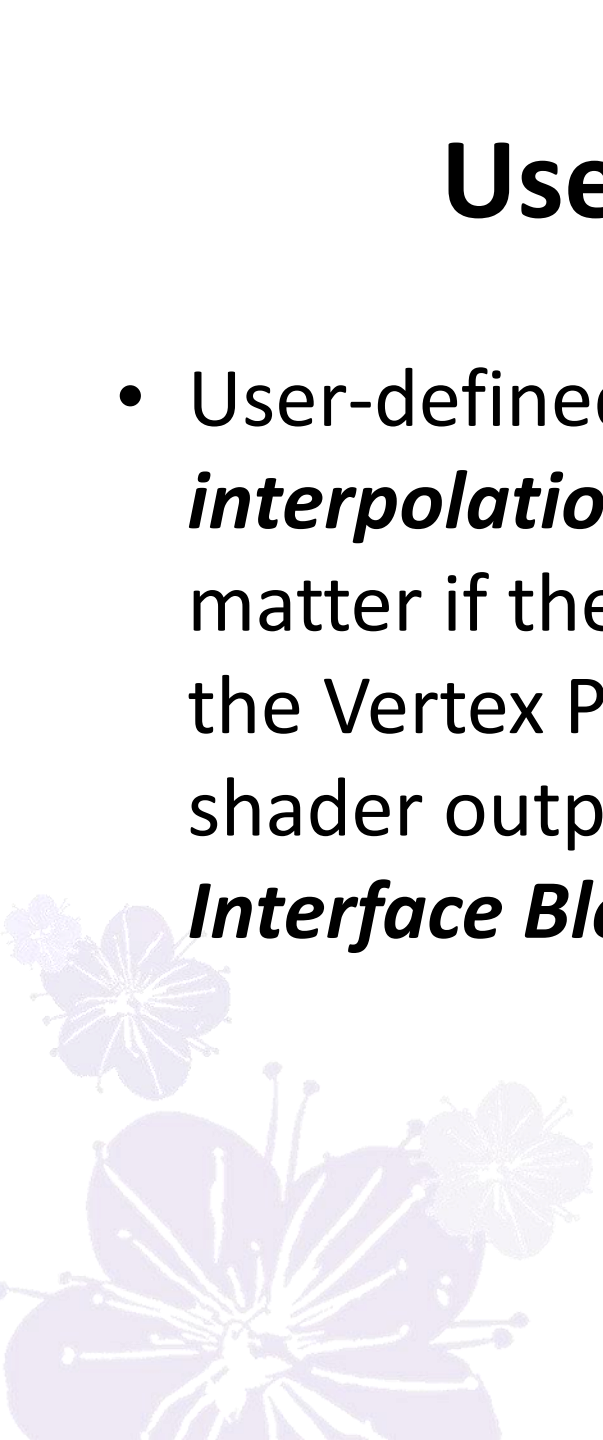
- ***gl_ClipDistance[]***
  - Allows the shader to set the ***distance from the vertex to each user-defined clipping half-space***
  - A non-negative distance means that the vertex is inside/behind the clip plane, and a negative distance means it is outside/in front of the clip plane. Each element in the array is one clip plane
  - In order to use this variable, the user must manually ***redeclare it with an explicit size***

gl_PerVertex

- ***gl_ClipDistance[]***
  - Clipping is also a complicated topic. We will introduce this to you later in a dedicated section

User-defined outputs

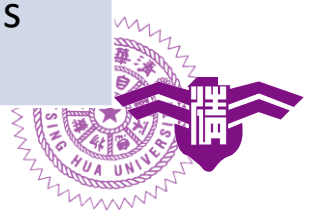
- User-defined output variables can have ***interpolation qualifiers*** (though these only matter if the output is being passed directly to the Vertex Post-Processing stage). Vertex shader outputs can also be aggregated into ***Interface Blocks***



Interpolation Qualifiers

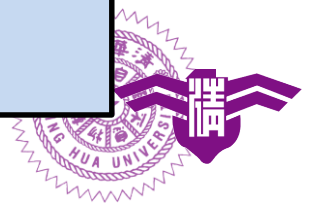
- Interpolation qualifiers control how interpolation of values happens across a triangle or other primitive in rasterizer

Qualifier Name	Description
flat	The value will not be interpolated. The value given to the fragment shader is the value from the Provoking Vertex for that primitive.
noperspective	The value will be linearly interpolated in window-space. This is usually not what you want, but it can have its uses.
smooth	The value will be interpolated in a perspective-correct fashion. This is the **default** if no qualifier is present.

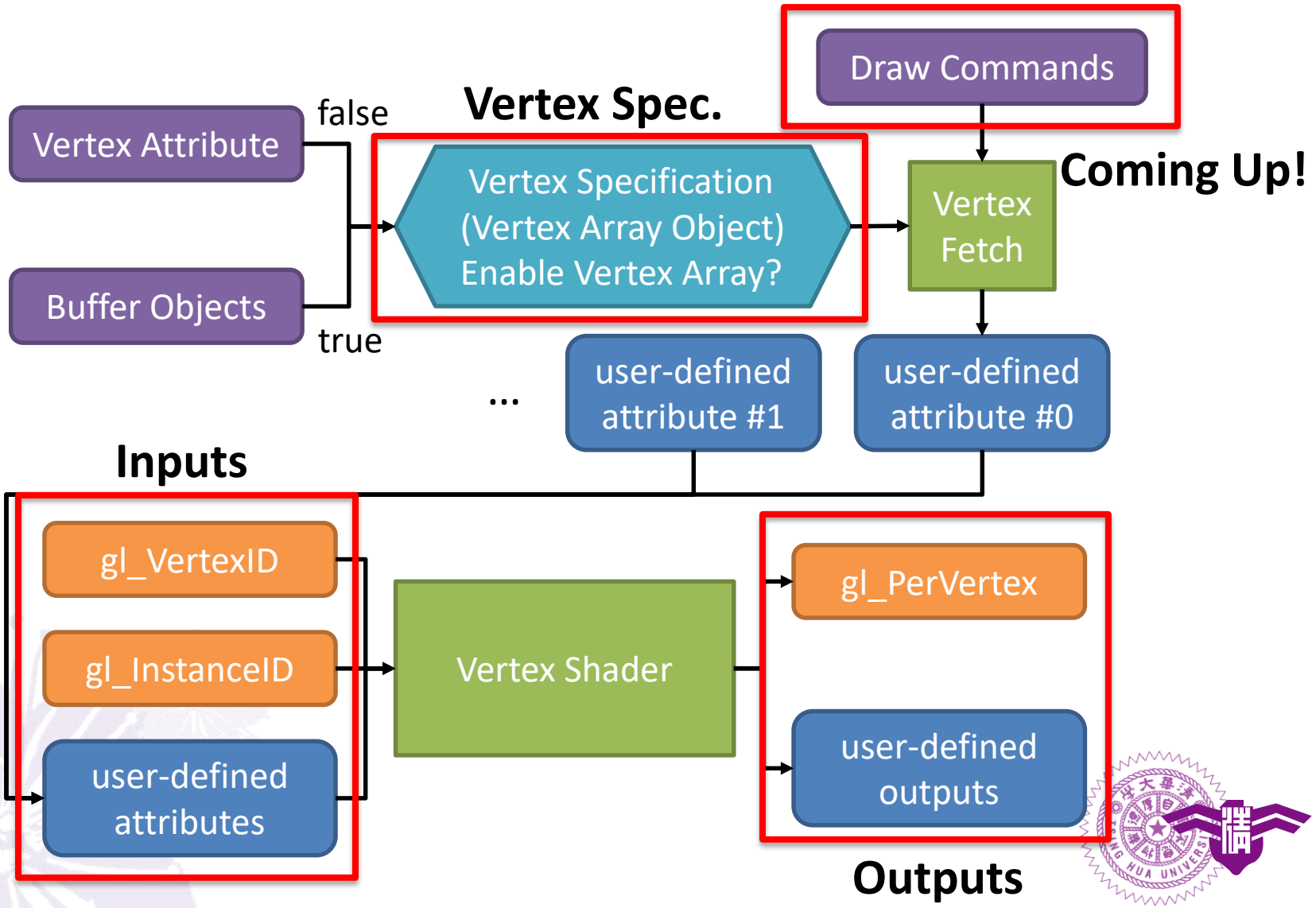


User-defined outputs

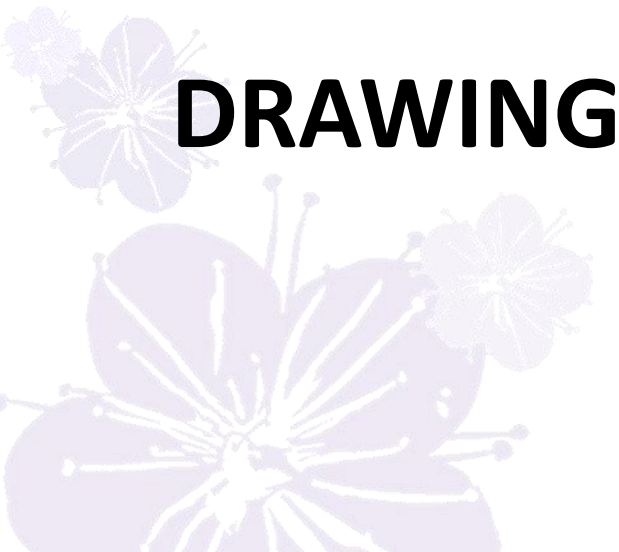
```
#version 410 core

layout (location = 0) in vec4 vertex;
layout (location = 1) in vec3 color;

out VertexData // output interface block
{
    vec4 vertex; // an output variable
    smooth vec3 color; // smooth interpolation (default)
    flat int vertexID; // flat interpolation
} vertexData;

void main(void)
{
    vertexData.vertex = vertex; // write to an output variable
    vertexData.color = color;
    vertexData.vertexID = gl_VertexID;
    gl_Position = vertex;
}
```



Vertex Shader Overview



DRAWING COMMANDS



Drawing Commands

- Drawing commands start ***vertex rendering***: the process of taking vertex data specified in ***vertex array object*** and rendering one or more ***primitives*** with this vertex data
- Non-indexed vs. indexed
- Instanced
- Direct vs. indirect

Basic Drawing

- **glDrawArrays**
  - Non-indexed drawing command
  - Vertices are issued ***in order***. Vertex data stored in buffers is simply fed to the vertex shader in the order that ***it appears in the buffer***
- **glDrawElements**
  - Indexed drawing command
  - Includes an ***indirection*** step that treats the data in each of the buffers as an array, and uses ***another index array to index into them***
  - Bind a buffer that contains the indices of the vertices to the **GL_ELEMENT_ARRAY_BUFFER** target

glDrawArrays

void glDrawArrays(GLenum mode, GLint first, GLsizei count);

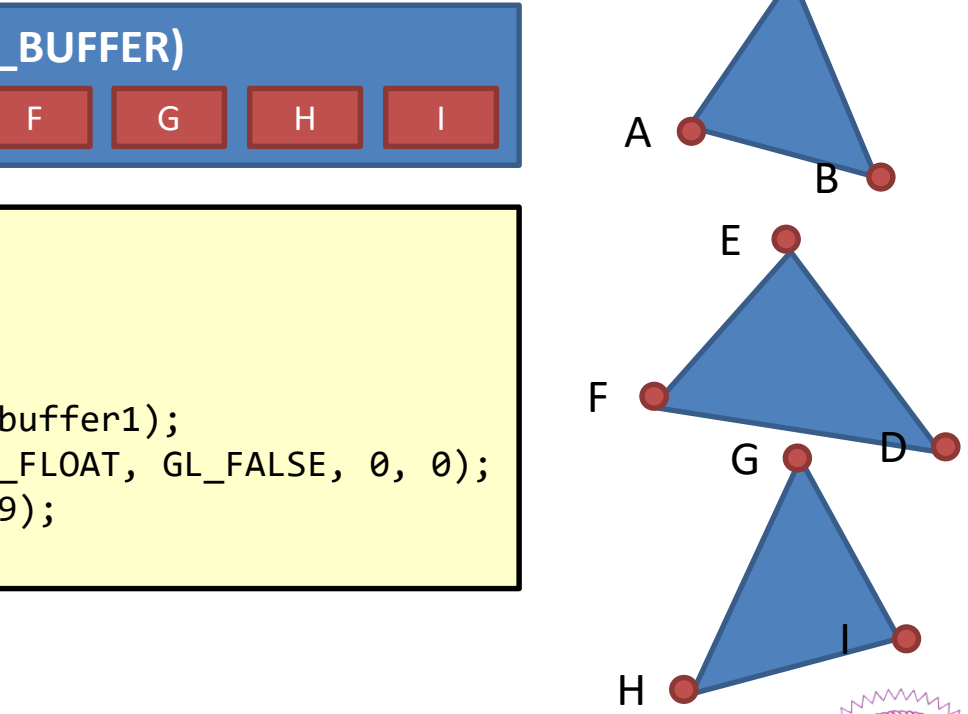
- **Description**: **glDrawArrays** constructs a sequence of geometric primitives using array elements starting at **first** and ending at **first + count – 1** of each enabled array. **mode** specifies what kinds of primitives are constructed

Mode for Drawing Commands

- GL_POINTS
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_LINES
- GL_LINE_STRIP_ADJACENCY
- GL_LINES_ADJACENCY
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_TRIANGLES
- GL_TRIANGLE_STRIP_ADJACENCY
- GL_TRIANGLES_ADJACENCY
- GL_PATCHES

Basic Drawing

Any Volunteers?




```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, buffer1);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, 9);
}
```

glDrawElements

void glDrawElements(GLenum mode, GLsizei count,
*GLenum type, const GLvoid *offset);*

- **Description**: **glDrawElemets** constructs a sequence of geometric primitives using **count** indices starting from **offset** bytes in the buffer bound to **GL_ELEMENT_ARRAY_BUFFER** target. **type** specifies the data type of the indices in the buffer. **mode** specifies what kinds of primitives are constructed

Create and Use Element Array

```
GLuint buffer;
static const int indices[] = { ... };

// Allocate and initialize a buffer object
glGenBuffers(1, &buffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(indices), indices, GL_STATIC_DRAW);

// Bind the element array buffer and use glDrawElements()
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glDrawElements(GL_TRIANGLES, n, GL_UNSIGNED_INT, 0);
```

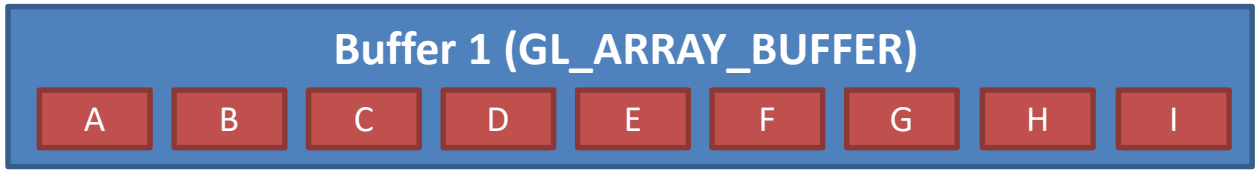
Type for Element Arrays

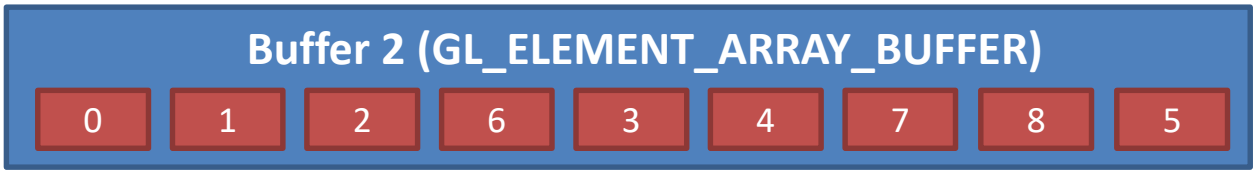
- Analyze your data for possible index range to choose the appropriate data type, saving both CPU and GPU memory

Name	Size in Bytes	Possible Values
GL_UNSIGNED_BYTE	1	[0, 255]
GL_UNSIGNED_SHORT	2	[0, 65535]
GL_UNSIGNED_INT	4	[0, 2147483647]

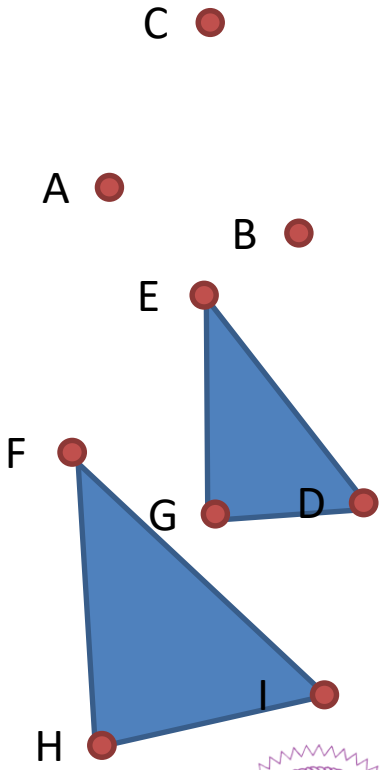
Basic Drawing

Any Volunteers?






```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, buffer1);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer2);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 12);
}
```



Base Index

- **glDrawElementsBaseVertex**
  - Index is offset by a base vertex value

*void glDrawElementsBaseVertex(GLenum mode, GLsizei count, GLenum type, const GLvoid *offset, GLint basevertex);*

- **Description**: fetch the vertex index from the buffer bound to the **GL_ELEMENT_ARRAY_BUFFER** and then add **basevertex** to it before it is used to index into the array of vertices

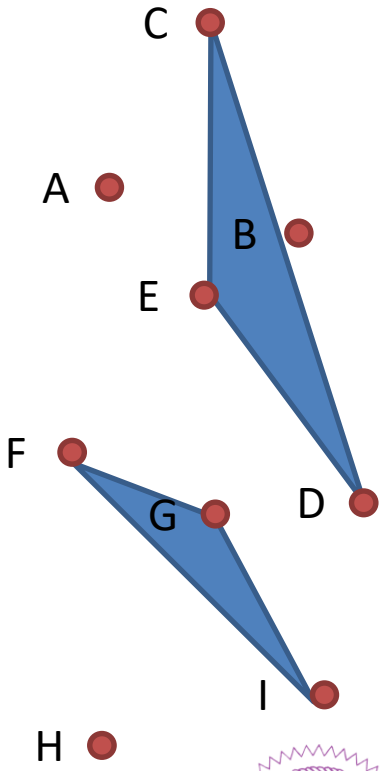
Base Index

Any Volunteers?

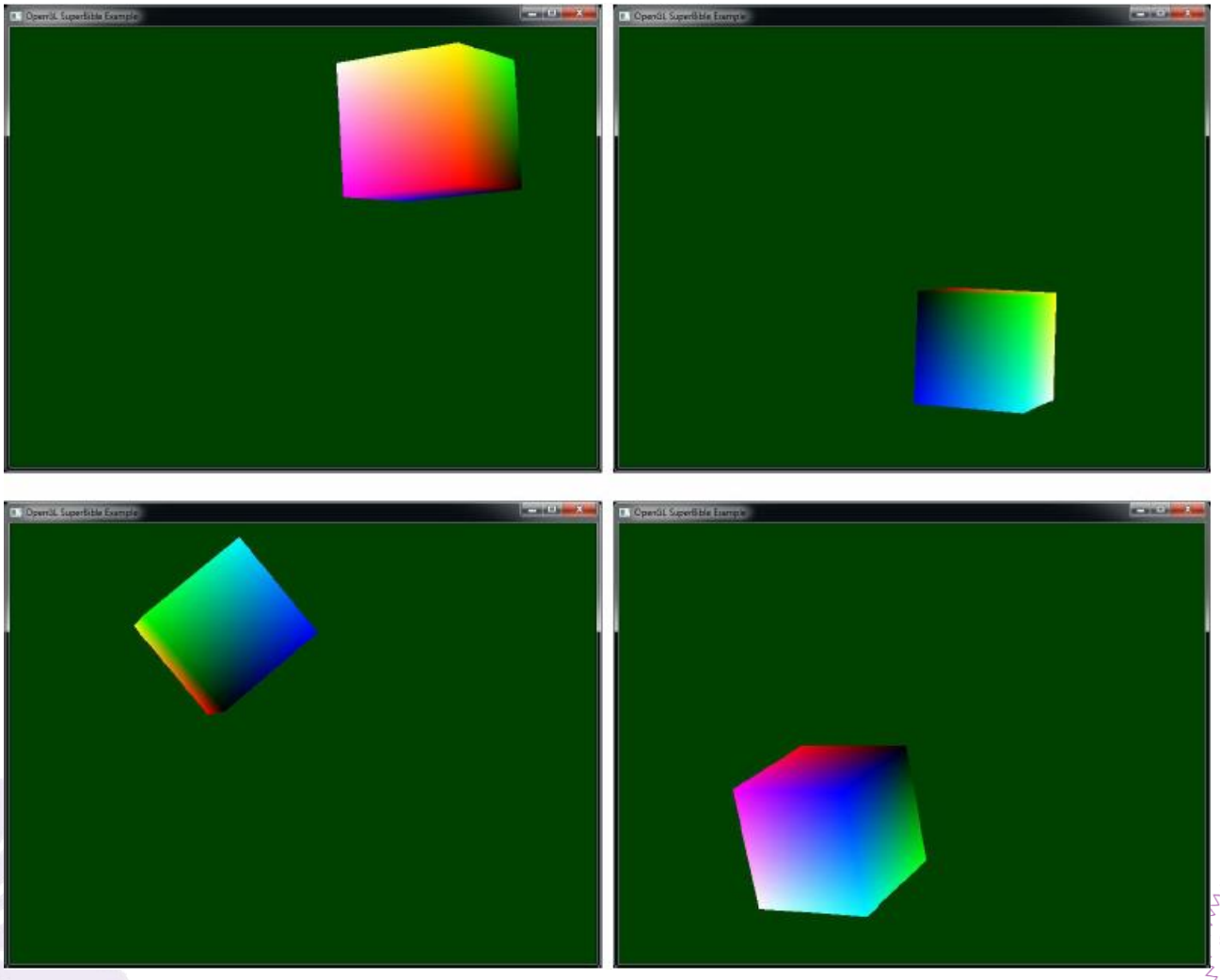
Buffer 1 (GL_ARRAY_BUFFER)								
A	B	C	D	E	F	G	H	I

Buffer 2 (GL_ELEMENT_ARRAY_BUFFER)								
0	1	2	6	3	4	7	8	5

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, buffer1);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer2);
    glDrawElementsBaseVertex(GL_TRIANGLES, 6,
        GL_UNSIGNED_INT, 0, 2);
}
```



Spinning Cube



Code: Variables

```
GLuint          vao;
GLuint          program;
GLuint          buffer;
GLint           mv_location;
GLint           proj_location;
float           aspect;
glm::mat4       proj_matrix;
```

Code: Vertex Shader

```
#version 410 core

in vec4 position;
out VS_OUT
{
    vec4 color;
} vs_out;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
    gl_Position = proj_matrix * mv_matrix * position;
    vs_out.color = position * 2.0 + vec4(0.5, 0.5, 0.5, 0.0);
}
```

Code: Fragment Shader

```
#version 410 core

in VS_OUT
{
    vec4 color;
} fs_in;

out vec4 color;

void main(void)
{
  color = fs_in.color;
}
```

Code: Buffer Data

```
static const GLfloat vertex_positions[] =
{
    -0.25f, -0.25f, -0.25f,
    -0.25f, 0.25f, -0.25f,
    0.25f, -0.25f, -0.25f,
    0.25f, 0.25f, -0.25f,
    0.25f, -0.25f, 0.25f,
    0.25f, 0.25f, 0.25f,
    -0.25f, -0.25f, 0.25f,
    -0.25f, 0.25f, 0.25f,
};

static const GLushort vertex_indices[] =
{
    0, 1, 2, 2, 1, 3,
    2, 3, 4, 4, 3, 5,
    4, 5, 6, 6, 5, 7,
    6, 7, 0, 0, 7, 1,
    6, 0, 2, 2, 4, 6,
    7, 5, 3, 7, 3, 1
};
```

Code: Set Up Buffer

```
glGenBuffers(1, &position_buffer);
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_positions),
vertex_positions, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

glGenBuffers(1, &index_buffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertex_indices),
vertex_indices, GL_STATIC_DRAW);
```

Code: Reshape Function

```
glm::mat4 proj_matrix;

void OnReshape(int w, int h)
{
  aspect = (float)w / (float)f;

  //should use radius in glm!!
  proj_matrix = glm::perspective(deg2rad(50.0f),
                aspect, 0.1f,1000.0f);

}
```

Code: Display Function

```
void My_Display()
{
    float f = (float)currentTime * 0.3f;
    glm::mat4 Identy_Init(1.0);

    glm::mat4 mv_matrix =
    glm::translate(Identy_Init, glm:: vec3(0.0f, 0.0f, -4.0f));

    mv_matrix = glm::translate(mv_matrix, glm::vec3(sinf(2.1f
*f)*0.5f, cosf(1.7f * f)*0.5f, sinf(1.3f * f)*cosf(1.5f*f)*2.0f));

    mv_matrix = glm::rotate(mv_matrix, deg2rad(currentTime*45.0f),
                                    glm::vec3(0.0f, 1.0f, 0.0f));

    mv_matrix = glm::rotate(mv_matrix, deg2rad(currentTime*81.0f),
                                    glm::vec3(1.0f, 0.0f, 0.0f));
```

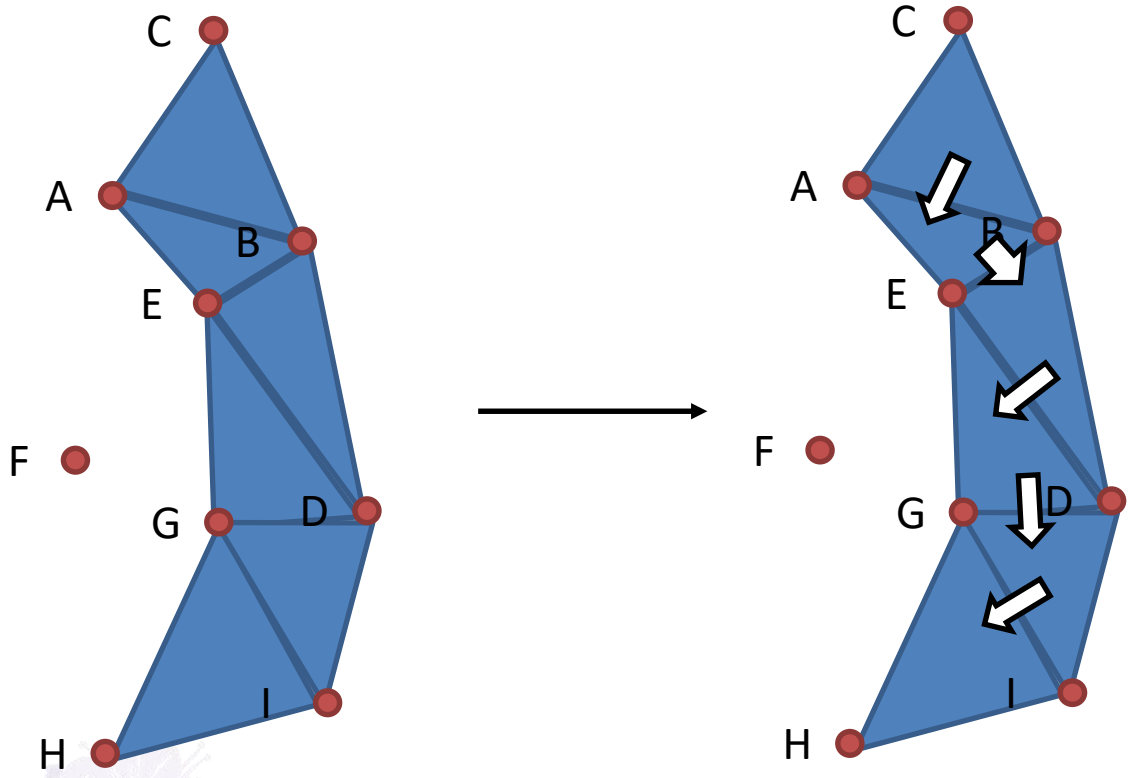
Code: Display Function (Cont'd)

```
    // Clear the framebuffer with dark green
    static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, green);
    // Activate our program
    glUseProgram(program);
    // Set the model-view and projection matrices
    glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
    glUniformMatrix4fv(proj_location, 1, GL_FALSE, proj_matrix);
    // Draw 6 faces of 2 triangles of 3 vertices each = 36 vertices
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
}
```

Drawing Triangle Strips

- Many tools are available that "***stripify***" a geometry
- The idea is that by taking "***triangle soup***," a large collection of unconnected triangles, attempt to merge it into a set of triangle strips
- Each individual triangle is represented by ***1 vertex instead of 3*** (except for the first triangle)
- By converting the geometry from triangle soup to triangle strips, there is less geometry data to process, and the system should run faster
- If the tool does a good job and produces a small number of long strips containing many triangles each, this generally works well

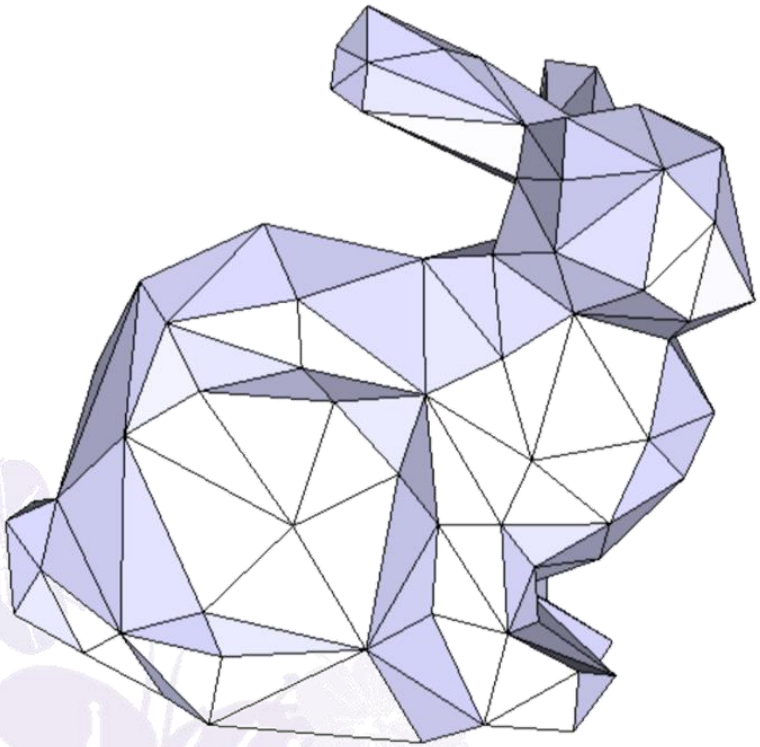
Drawing Triangle Strips

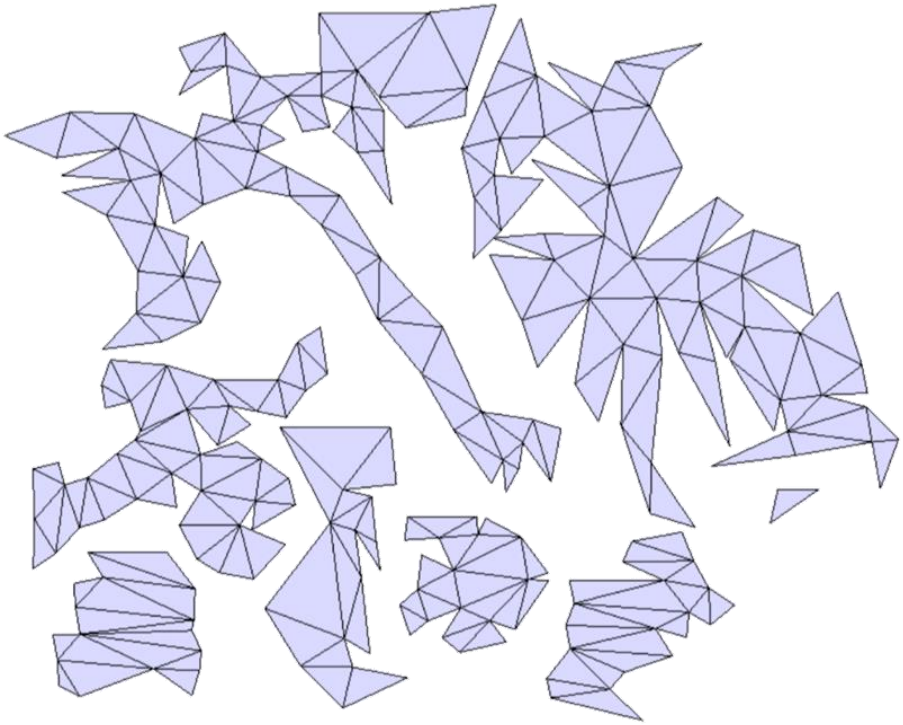


GL_TRIANGLES:
CABBAEBEDDEGDGIIGH

GL_TRIANGLE_STRIP:
CABEDGIH

Triangle Strips: Example





Primitive Restart

- Without **GL_PRIMITIVE_RESTART**, only 1 triangle strip can be rendered by a single draw command of **glDrawArrays** or **glDrawElements**
- OpenGL checks for a special index value; whenever it comes across it, OpenGL ends the current strip and starts a new one with the next vertex
- **GL_PRIMITIVE_RESTART** applies to these modes:
  - GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_LINE_STRIP, GL_LINE_LOOP
- Ignored in non-indexed drawing calls

Primitive Restart

void glPrimitiveRestartIndex(GLuint index);

- **Description**: specify the index value to be interpreted as the restart index by **index**
- **GL_PRIMITIVE_RESTART** should be enabled to use this feature (default disabled)

```
glEnable(GL_PRIMITIVE_RESTART);
glPrimitiveRestartIndex(restartIdx); // default = 0
glDrawElements(GL_TRIANGLE_STRIP, n, GL_UNSIGNED_INT, 0);
glDisable(GL_PRIMITIVE_RESTART);
```

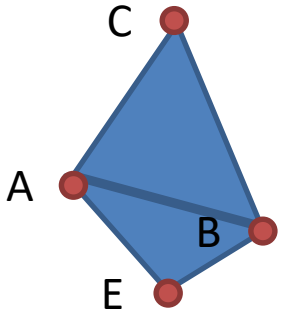
Primitive Restart

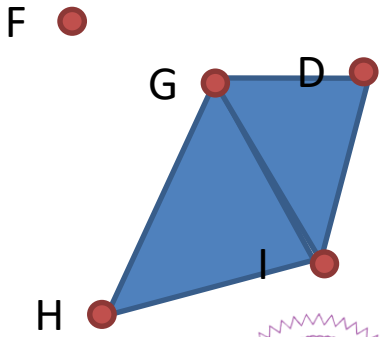
Buffer 1 (GL_ARRAY_BUFFER)								
A	B	C	D	E	F	G	H	I

Buffer 2 (GL_ELEMENT_ARRAY_BUFFER)								
2	0	1	4	9	3	6	8	7

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, buffer1);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer2);
    glEnable(GL_PRIMITIVE_RESTART);
    glPrimitiveRestartIndex(9);
    glDrawElements(GL_TRIANGLE_STRIP, 9,
        GL_UNSIGNED_INT, 0);
}
```

Any Volunteers?

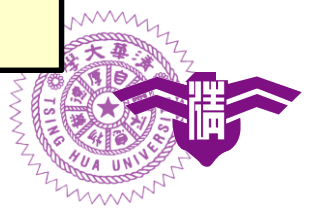




Instancing

- There will probably be times when you want to draw the same object many times
- A field of grass?
- There could be thousands of copies of identical sets of geometry, modified only slightly from instance to instance
- Something like this?

```
glBindVertexArray(grass_vao);
for (int n = 0; n < number_of_blades_of_grass; n++)
{
    SetupGrassBladeParameters();
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 6);
}
```



Instancing (Cont'd)

- There could be thousands, millions of grass!
- Rendering each is cheap in terms of GPU cost
- The system is likely to spend most of its time sending commands to OpenGL
- Instanced rendering is a method provided by OpenGL to draw many copies of the same geometry with a single function call without extra system overhead
- **glDrawArraysInstanced**
  - Instanced version of glDrawArrays
- **glDrawElementsInstanced**
  - Instanced version of glDrawElements

Instancing: APIs

void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei instancecount);

- **Description**: use **mode**, **first** and **count** to render **instancecount** times

*void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const GLvoid *offset, GLsizei instancecount);*

- **Description**: use **mode**, **count**, **type** and **offset** to render **instancecount** times

Instancing: Example

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, grass_vertex_buffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, grass_index_buffer);
    glDrawElementsInstanced(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0, 10000);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;

void main(void)
{
    gl_Position = vec4(vertex, 1.0);
}
```

10,000 same grass overlapped with each other!

Instancing: Concepts

- How is the instanced version different from the loop one?

1. You cannot change any state between each instance rendering, while it is possible in the loop version
2. The instanced version runs faster because OpenGL only fetch state/generate command once
3. **gl_InstanceID** is always 0 in the loop version; while in the instanced version, its range is [0, instancecount)

Per-Instance Data

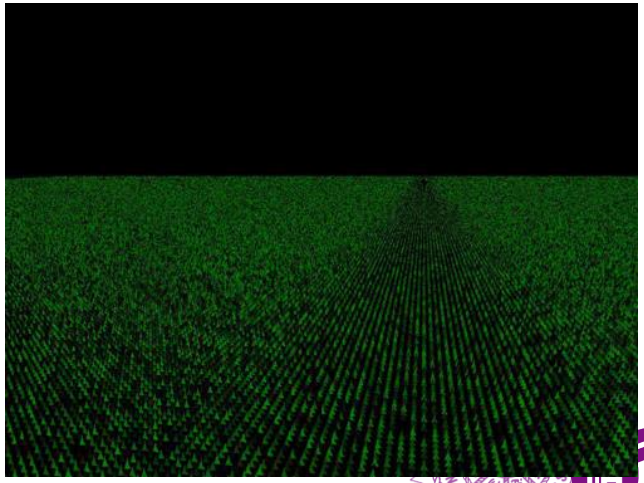
- So far, the only difference in each instance is the **gl_InstanceID** input variable
- How to render grass with different position/ length/orientation?

1. Use **gl_InstanceID** for procedural parameter generation
2. Use **gl_InstanceID** to index into uniform array variables
3. Use ***vertex attribute divisors***

Per-Instance Data (1)

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, grass_vertex_buffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, grass_index_buffer);
    glDrawElementsInstanced(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0, 10000);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;

void main(void)
{
    int id = gl_InstanceID;
    vec3 offset = vec3(id/100, mod(id, 100), 0);
    gl_Position = vec4(vertex + offset, 1.0);
}
```



Per-Instance Data (2)

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, grass_vertex_buffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, grass_index_buffer);
    glDrawElementsInstanced(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0, 10000);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;
uniform MyUniformBlock {
    vec3 offset[10000];
};

void main(void)
{
    gl_Position = vec4(vertex +
        offset[gl_InstanceID], 1.0);
}
```

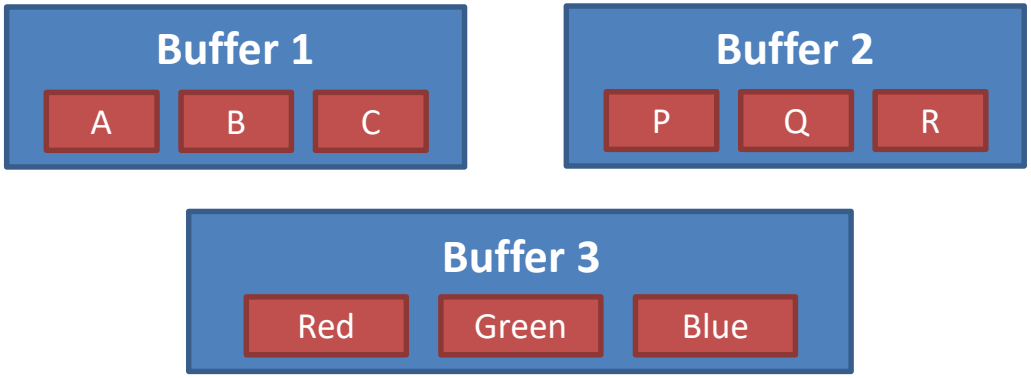
Per-Instance Data (3)

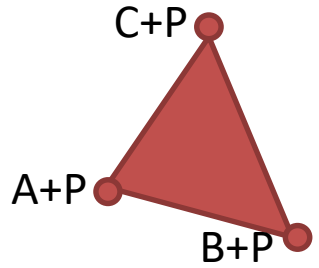
- Normally, the vertex attributes would be fetched per-vertex
- To make OpenGL read attributes once per instance, use **glVertexAttribDivisor**

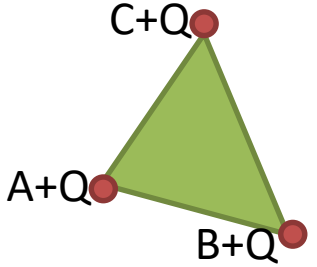
void glVertexAttribDivisor(GLuint index, GLuint divisor);

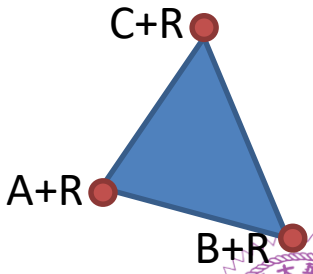
- **Description**: set the vertex fetch frequency of the vertex attribute specified by **index**. If **divisor** is zero, attribute is fetched per-vertex; If **divisor** is positive, attribute is fetched once every **divisor** instances

Per-Instance Data (3)










```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(vertex);
    glEnableVertexAttribArray(offset);
    glEnableVertexAttribArray(color);
    ... // Bind 3 buffers
    glVertexAttribDivisor(offset, 1);
    glVertexAttribDivisor(color, 1);
    glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 3);
}
```

Per-Instance Data (3)

```
void render()
{
    glBindVertexArray(vao);
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glVertexAttribDivisor(1, 1);
    glBindBuffer(GL_ARRAY_BUFFER, grass_vertex_buffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ARRAY_BUFFER, grass_offset_buffer);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, grass_index_buffer);
    glDrawElementsInstanced(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0, 10000);
}
```

```
#version 410 core

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 offset;

void main(void)
{
    gl_Position = vec4(vertex + offset, 1.0);
}
```

Base Instance

- You can add an offset to per-instance data index using BaseInstance versions
- **glDrawArraysInstancedBaseInstance**
- **glDrawElementsInstancedBaseVertexBaseInstance**
- Setting base instance doesn't effect **gl_InstanceID**; Base instance is used to offset per-instance vertex attribute fetches (set by **glVertexAttribDivisor** calls)
- $Index = \left\lfloor \frac{InstanceID}{Divisor} \right\rfloor + BaseInstance$

Base Instance

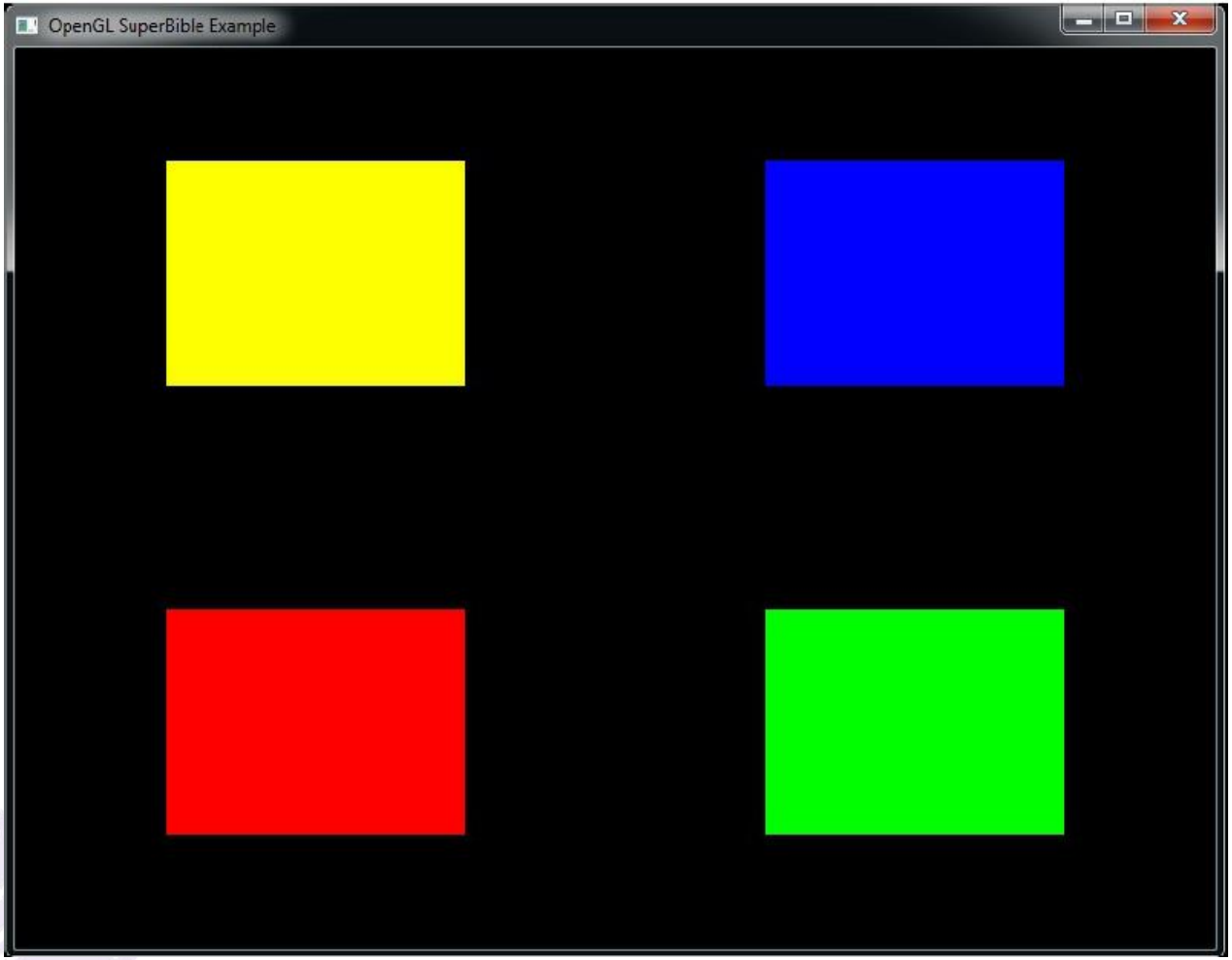
void glDrawArraysInstancedBaseInstance(GLenum mode, GLint first, GLsizei count, GLsizei instancecount, GLint baseinstance);

- **Description**: same as **glDrawArraysInstanced** with additional **baseinstance** parameter

*void glDrawElementsInstancedBaseVertexBaseInstance(GLenum mode, GLsizei count, GLenum type, const GLvoid *offset, GLsizei instancecount, GLint basevertex, GLint baseinstance);*

- **Description**: same as **glDrawElementsInstanced** with **basevertex** and **baseinstance** parameter

Instanced Rendering



Code: Vertex Shader

```
#version 410

in vec4 position;
in vec4 instance_color;
in vec4 instance_position;

out Fragment
{
    vec4 color;
} fragment;

uniform mat4 mvp;

void main(void)
{
    gl_Position = mvp * (position + instance_position);
    fragment.color = instance_color;
}
```

Code: Fragment Shader

```
#version 410

in Fragment
{
    vec4 color;
} fragment;

out vec4 fragmentColor;

void main(void)
{
    fragmentColor = fragment.color;
}
```

Code: Buffer Data

```
static const GLfloat square_vertices[] =
{
    -1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f
};
static const GLfloat instance_colors[] =
{
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f
};
static const GLfloat instance_positions[] =
{
    -2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, 2.0f, 0.0f, 0.0f,
    -2.0f, 2.0f, 0.0f, 0.0f
};
```

Code: Set Up Buffer

```
GLuint offset = 0;
glGenVertexArrays(1, &square_vao);
glGenBuffers(1, &square_vbo);
glBindVertexArray(square_vao);
glBindBuffer(GL_ARRAY_BUFFER, square_vbo);

glBufferData(GL_ARRAY_BUFFER,
sizeof(square_vertices) +
sizeof(instance_colors) +
sizeof(instance_positions), NULL, GL_STATIC_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, offset,
sizeof(square_vertices),
square_vertices);
offset += sizeof(square_vertices);

glBufferSubData(GL_ARRAY_BUFFER, offset,
sizeof(instance_colors), instance_colors);
offset += sizeof(instance_colors);

glBufferSubData(GL_ARRAY_BUFFER, offset,
sizeof(instance_positions), instance_positions);
offset += sizeof(instance_positions);
```

Code: Set Up Attribute

```
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0,
0);

glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
(GLvoid *)sizeof(square_vertices));

glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0,
(GLvoid *)(sizeof(square_vertices) +
sizeof(instance_colors)));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glVertexAttribDivisor(1, 1);
glVertexAttribDivisor(2, 1);
```

Code: Display Function

```
void My_Display()
{
    static const GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f };

    glClearBufferfv(GL_COLOR, 0, black);

    glUseProgram(instancingProg);
    glBindVertexArray(square_vao);
    glDrawArraysInstanced(GL_TRIANGLE_FAN, 0, 4, 4);
}
```

Indirect Draws

- So far, we have covered only ***direct*** drawing commands. In these commands, we pass the parameters of the drawing command
- ***Indirect*** draws allow the parameters of each drawing command to be stored in a buffer object
- As a consequence, at the time that your application calls the drawing command, it doesn't actually need to know those parameters—only the location in the buffer where the parameters are stored

Indirect Draws

- Generate the parameters for a drawing command ahead of time, even offline, then load them into a buffer and send them to OpenGL
- Use OpenGL to generate the parameters at runtime by executing a shader that stores those parameters in a buffer object
- Use many threads on the CPU to generate the parameters for drawing commands

Indirect Draws

- **glDrawArraysIndirect**
  - The indirect version of **glDrawArraysInstancedBaseInstance**
- **glDrawElementsIndirect**
  - The indirect version of **glDrawElementsInstancedBaseVertexBaseInstance**
- Store the parameters of the draw commands in a buffer bound to **GL_DRAW_INDIRECT_BUFFER** target in pre-defined format

Indirect Draws

*void glDrawArraysIndirect(GLenum mode, const GLvoid *offset);*

- **Description**: The indirect version of **glDrawArraysInstancedBaseInstance**, starting from **offset** in the buffer bound to **GL_INDIRECT_DRAW_BUFFER** taget

*void glDrawElementsIndirect(GLenum mode, Glenum type, const GLvoid *offset);*

- **Description**: The indirect version of **glDrawElementsInstancedBaseVertexBaseInstance**, starting from **offset** in the buffer bound to **GL_INDIRECT_DRAW_BUFFER** taget

Indirect Draws

*void glMultiDrawArraysIndirect(GLenum mode, const GLvoid *offset, GLsizei drawcount, GLsizei stride);*

- **Description**: The indirect version of **glDrawArraysInstancedBaseInstance**, starting from **offset** in the buffer bound to **GL_INDIRECT_DRAW_BUFFER** taget, issue **drawcount** commands in order. Each command is offset by **stride** bytes. If **stride** is zero, the indirect draw buffer is considered to be tightly-packed

Indirect Draws

*void glMultiDrawElementsIndirect(GLenum mode, Glenum type, const GLvoid *offset , GLsizei drawcount, GLsizei stride);*

- **Description**: The indirect version of **glDrawElementsInstancedBaseVertexBaseInstance**, starting from **offset** in the buffer bound to **GL_INDIRECT_DRAW_BUFFER** taget, issue **drawcount** commands in order. Each command is offset by **stride** bytes. If **stride** is zero, the indirect draw buffer is considered to be tightly-packed

Indirect Draws

```
typedef struct
{
    GLuint vertexCount;
    GLuint instanceCount;
    GLuint firstVertex;
    GLuint baseInstance;
} DrawArraysIndirectCommand;

DrawArraysIndirectCommand
buffer[];
```

DrawArraysIndirect

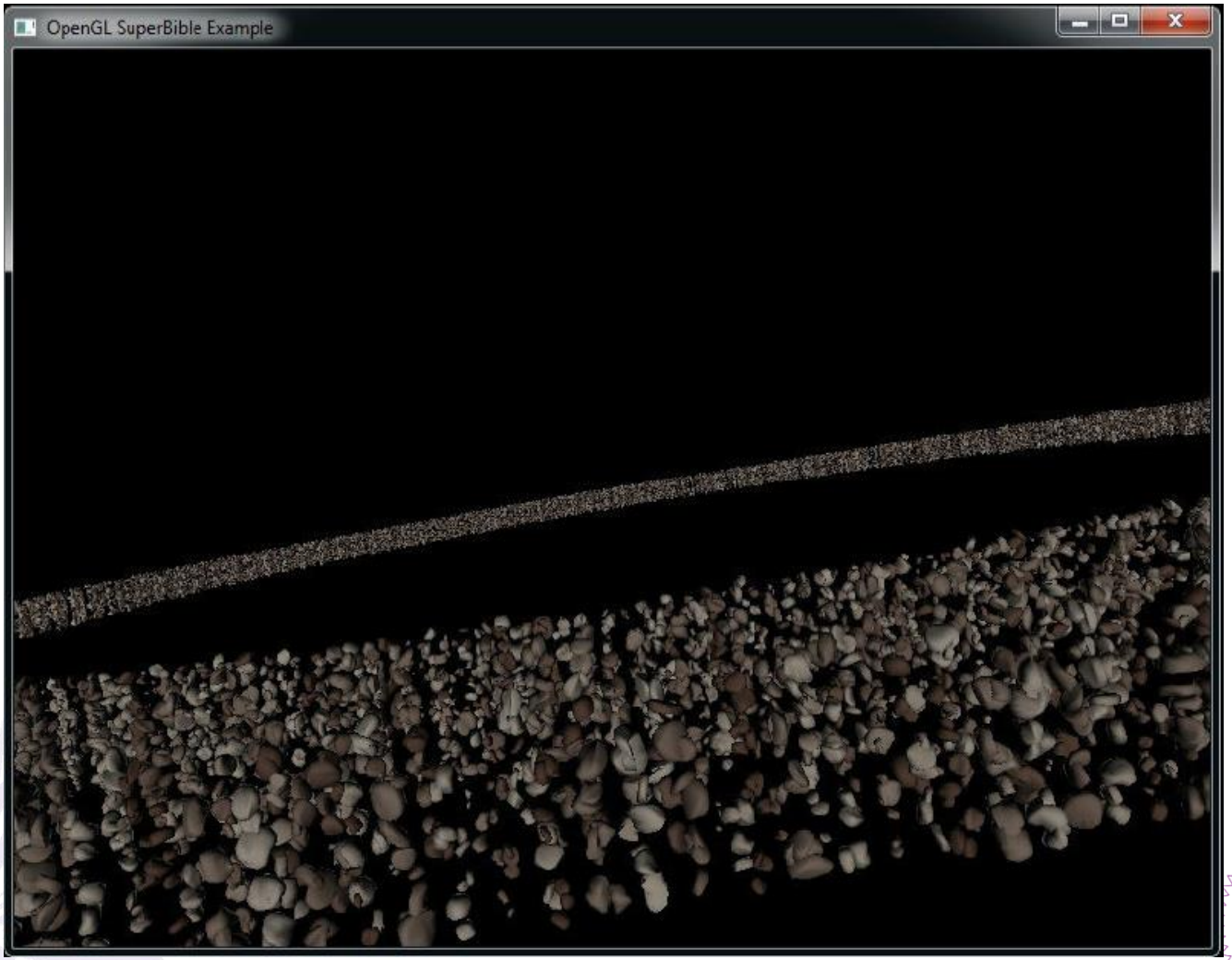
```
typedef struct
{
    GLuint vertexCount;
    GLuint instanceCount;
    GLuint firstIndex;
    GLint baseVertex;
    GLuint baseInstance;
} DrawElementsIndirectCommand;

DrawElementsIndirectCommand
buffer[];
```

DrawElemetsIndirect

Indirect Draws

```
GLuint buffer;
GLuint commandList[num_commands * 5];

for(int i = 0; i < num_commands; i++) {
    commandList[i * 5 + 0] = vertexCount;
    commandList[i * 5 + 1] = instanceCount;
    commandList[i * 5 + 2] = firstIndex;
    ((GLint *)commandList)[i * 5 + 3] = baseVertex;
    commandList[i * 5 + 4] = baseInstance;
}

// Allocate and initialize a buffer object
glGenBuffers(1, &buffer);
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, buffer);
glBufferData(GL_DRAW_INDIRECT_BUFFER,
    sizeof(GLuint) * num_commands * 5, commandList, GL_STATIC_DRAW);

// Bind the draw indirect array buffer and glMultiDrawElementsIndirect()
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, buffer);
glMultiDrawElementsIndirect(GL_TRIANGLES, GL_UNSIGNED_INT,
                            0, num_commands, 0);
```

Asteroid Rendering



Code: Vertex Shader

```
#version 410
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
layout (location = 10) in uint draw_id;

out VS_OUT
{
    vec3 normal;
    vec4 color;
} vs_out;

uniform float time = 0.0;
uniform mat4 view_matrix;
uniform mat4 proj_matrix;
uniform mat4 viewproj_matrix;

const vec4 color0 = vec4(0.29, 0.21, 0.18, 1.0);
const vec4 color1 = vec4(0.58, 0.55, 0.51, 1.0);
```

Code: Vertex Shader (Cont'd)

```
void main(void)
{
    mat4 m1;
    mat4 m2;
    mat4 m;
    float t = time * 0.1;
    float f = float(draw_id) / 30.0;
    float st = sin(t * 0.5 + f * 5.0);
    float ct = cos(t * 0.5 + f * 5.0);
    float j = fract(f);
    float d = cos(j * 3.14159);
    // Rotate around Y
    m[0] = vec4(ct, 0.0, st, 0.0);
    m[1] = vec4(0.0, 1.0, 0.0, 0.0);
    m[2] = vec4(-st, 0.0, ct, 0.0);
    m[3] = vec4(0.0, 0.0, 0.0, 1.0);
    // Translate in the XZ plane
    m1[0] = vec4(1.0, 0.0, 0.0, 0.0);
    m1[1] = vec4(0.0, 1.0, 0.0, 0.0);
    m1[2] = vec4(0.0, 0.0, 1.0, 0.0);
    m1[3] = vec4(260.0 + 30.0 * d, 5.0 * sin(f * 123.123), 0.0, 1.0);
    m = m * m1;
```

Code: Vertex Shader (Cont'd)

```
    // Rotate around X
    st = sin(t * 2.1 * (600.0 + f) * 0.01);
    ct = cos(t * 2.1 * (600.0 + f) * 0.01);
    m1[0] = vec4(ct, st, 0.0, 0.0);
    m1[1] = vec4(-st, ct, 0.0, 0.0);
    m1[2] = vec4(0.0, 0.0, 1.0, 0.0);
    m1[3] = vec4(0.0, 0.0, 0.0, 1.0);
    m = m * m1;
    // Rotate around Z
    st = sin(t * 1.7 * (700.0 + f) * 0.01);
    ct = cos(t * 1.7 * (700.0 + f) * 0.01);
    m1[0] = vec4(1.0, 0.0, 0.0, 0.0);
    m1[1] = vec4(0.0, ct, st, 0.0);
    m1[2] = vec4(0.0, -st, ct, 0.0);
    m1[3] = vec4(0.0, 0.0, 0.0, 1.0);
    m = m * m1;
    // Non-uniform scale
    float f1 = 0.65 + cos(f * 1.1) * 0.2;
    float f2 = 0.65 + cos(f * 1.1) * 0.2;
    float f3 = 0.65 + cos(f * 1.3) * 0.2;
    m1[0] = vec4(f1, 0.0, 0.0, 0.0);
    m1[1] = vec4(0.0, f2, 0.0, 0.0);
    m1[2] = vec4(0.0, 0.0, f3, 0.0);
    m1[3] = vec4(0.0, 0.0, 0.0, 1.0);
    m = m * m1;
    gl_Position = viewproj_matrix * m * position;
    vs_out.normal = mat3(view_matrix * m) * normal;
    vs_out.color = mix(color0, color1, fract(j * 313.431));
}
```

Code: Set Up Buffer

```
object.load("media/objects/asteroids.sbm");
glGenBuffers(1, &indirect_draw_buffer);
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, indirect_draw_buffer);
glBufferData(GL_DRAW_INDIRECT_BUFFER,
NUM_DRAWS * sizeof(DrawArraysIndirectCommand), NULL, GL_STATIC_DRAW);

DrawArraysIndirectCommand * cmd = (DrawArraysIndirectCommand *)
glMapBufferRange(GL_DRAW_INDIRECT_BUFFER, 0,
NUM_DRAWS * sizeof(DrawArraysIndirectCommand),
GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);

for (i = 0; i < NUM_DRAWS; i++)
{
    object.get_sub_object_info(i % object.get_sub_object_count(),
        cmd[i].first,
        cmd[i].count);
    cmd[i].primCount = 1;
    cmd[i].baseInstance = i;
}
glUnmapBuffer(GL_DRAW_INDIRECT_BUFFER);
```

Code: Set Up Buffer

```
glBindVertexArray(object.get_vao());
glGenBuffers(1, &draw_index_buffer);
glBindBuffer(GL_ARRAY_BUFFER, draw_index_buffer);
glBufferData(GL_ARRAY_BUFFER, NUM_DRAWS * sizeof(GLuint), NULL, GL_STATIC_DRAW);

GLuint * draw_index = (GLuint *)glMapBufferRange(GL_ARRAY_BUFFER, 0,
NUM_DRAWS * sizeof(GLuint), GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);

for (i = 0; i < NUM_DRAWS; i++)
{
    draw_index[i] = i;
}
glUnmapBuffer(GL_ARRAY_BUFFER);
glVertexAttribIPointer(10, 1, GL_UNSIGNED_INT, 0, NULL);
glVertexAttribDivisor(10, 1);
glEnableVertexAttribArray(10);
```

Code: Set Up Attribute

```
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
(GLvoid *)sizeof(square_vertices));

glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0,
(GLvoid *)(sizeof(square_vertices) +
sizeof(instance_colors)));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glVertexAttribDivisor(1, 1);
glVertexAttribDivisor(2, 1);
```

Code: Display Function

```
glBindVertexArray(object.get_vao());
if (mode == MODE_MULTIDRAW)
{
    glMultiDrawArraysIndirect(GL_TRIANGLES, NULL, NUM_DRAWS, 0);
}
else if (mode == MODE_SEPARATE_DRAWS)
{
    for (j = 0; j < NUM_DRAWS; j++)
    {
        GLuint first, count;
        object.get_sub_object_info(j % object.get_sub_object_count(),
            first, count);
        glDrawArraysInstancedBaseInstance(GL_TRIANGLES, first,
            count, 1, j);
    }
}
```