**Objective**

These labs will be used to practice the lessons and techniques learned in lecture. The labs themselves will not be graded, and don’t need to be turned in for credit at the end of lab. You can also work together to solve problems, however you are expected to know how the lab works, and will be tested on it the following day.

For each lab you will be given a few “samples”, working segments of code that do one specific thing and that thing only. It will be your job to use these, the information in your book, and what was taught in lecture to piece together a working version of the example. This will test your knowledge of the material as well as your critical thinking.

Included in this lab are the following:

* Example EXE of the final working program.
* Sample of loading a shader.
* Sample of implementing MSAA.

**Homework Requirements**

Features you’re expected to implement:

* The scene is properly displaying to the screen.
* Alt + Enter properly switch between full screen and windowed mode.
* F1, F2, and F3 switch from no multisampling anti-aliasing, 4x MSAA, and 8x MSAA, respectively.
* Shapes are drawn with colors, and outlines.
* Shapes move and bounce off sides of the screen.

**Homework Instructions**

**D3D App Class**

* You should probably create a class that handles all your d3d stuff, such as setup, rendering, updating, and shutdown. More hints on this are listed in the WindowsApp sample.
* Your application will need to load in and use both the screen2clip\_VS and basicColor\_PS shader files. Loading in a pixel shader is pretty similar to loading in a vertex shader.

**Shader Interaction**

* In this lab, the shader has already been created for you, however you’ll need to make sure you are passing it the data it’s expecting. You’ll need to create an InputLayout object that defines how our data is structured. This is the definition of your vertex structure, and will need to match the stuct you’re using for your verts, which is stored in SharedDefines.h. For this lab, our input layout will need:   
  a “POSITION” that is a DXGI\_FORMAT\_R32G32B32\_FLOAT,   
  a “COLOR” that is a DXGI\_FORMAT\_R32G32B32A32\_FLOAT,   
  and a “TEXCOORD” that is a  DXGI\_FORMAT\_R32\_UINT.
* Once we have this we will need to create our geometry and vertex buffer.
* The shader is built to handle up to 32 shapes in screen space, we’ll need to identify each vertex with an index, defining which shape each vert is a part of.
* Each shape is controlled by a matrix found in the SHAPE\_DATA namespace. You’ll need to update these matrices, then send them to the vertex shader via a constant buffer object, which you will need to create and update every frame.
* Don’t worry too much about what exactly the shader is doing right now (we’ll cover that more next class) just how it’s interacting with your code. Know that a matrix updates position, rotation, and scale, and the details about the shape data and InputLayout above.

**Constant Buffers**

* The structures for our constant buffers are stored in SharedDefines.h.
* Constant Buffers are how you’ll actually pass data from your code to your shaders. In this case we’ll have two constant buffers we need to pass to the shader.
* The first is the SCENE\_DATA, this contains a struct (SCENE\_INFO) that holds the dimensions of the backbuffer (a float2).
* The second one is the actual data for the objects SHAPE\_DATA, and contains an array of SHAPE\_INFO structures. Each structure contains a matrix for each shape (so we can move it) and a color for each shape
* **NOTE:**Typically each vertex buffer in a scene holds a single object (This will be the case for the remaning labs), so the constant buffer that we use with that object needs only one world matrix. That isn't the case with today's lab, we are rendering multiple objects with one vertex buffer, because of this, our SHAPE\_INFO constant buffer needs multiple world matricies. **This is only for this lab**. You'll need to make new constant buffers in future labs, so don't get too used to using the allShapes constant buffer.
* To actually send data over to the constant buffer we’ll need to do something similar to this in our C++ code:

D3D11\_MAPPED\_SUBRESOURCE edit;

immediateContext->Map(shapeData, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &edit);

memcpy(edit.pData, SHAPE\_DATA::allShapes, sizeof(SHAPE\_INFO) \* MAX\_SHAPES);

immediateContext->Unmap(shapeData, 0);

* This will pass along all the data in our shapeData constant buffer to vram so our shader can use it properly!

**Windows Application**

* You have been given a simple, basic windows application for reference. It’s suggested that you create an application class to house the majority of your code, then use an instance of this class in winMain, to keep things organized. The example is not setup like this.

**Creating the geometry**

* The geometry in the scene is a series of triangles and quads made at the origin of the screen, then line lists are used to draw the outlines after all the geometry.

**Moving the geometry**

* Each object will need its own matrix, which will be updated with an X and Y velocity, as well as a rotation. When the object reaches the edge of the screen, the object should “bounce” off of it. This can be done simply by reversing the applicable X or Y velocity.

**Alt + Enter to Full Screen**

* Your app should be able to switch from fullscreen to windowed and back with the press of Alt + Enter. This is relatively straight forward to setup, so a specific sample for it is not given.

**Multi-Sample Anti Aliasing (MSAA)**

* An example is included showing how this could be applied to a window. Again, it’s structured as part of the WinMain function, which shouldn’t be how you do it in your project.