 D3D11 Hardware Project

# Objective

Gain the knowledge & behaviors required to be successful in this course and beyond.

Your primary goal is to get to the end of this document. Once you do so you can relax a bit since you will be able to focus on other homework/studying as needed.

# Introduction

Welcome to the Direct3D11 Hardware Project. This is the only hardware assignment which holds your hand completely the entire way through! Future classes will demand the habits & foundation you develop right now! It is CRITICAL that you follow each and every step. ESPECIALLY THE READING!!!

# Pre-Requisites

* Strong C++ skills.
* Working knowledge of Geometry and Linear Algebra.
* A willingness to make mistakes and try, try again! You are here to learn after all!

“There are no secrets to success. It is the result of preparation, hard work, and learning from failure.”

-Colin Powell

“There is a way out of every box, a solution to every puzzle; it's just a matter of finding it.”

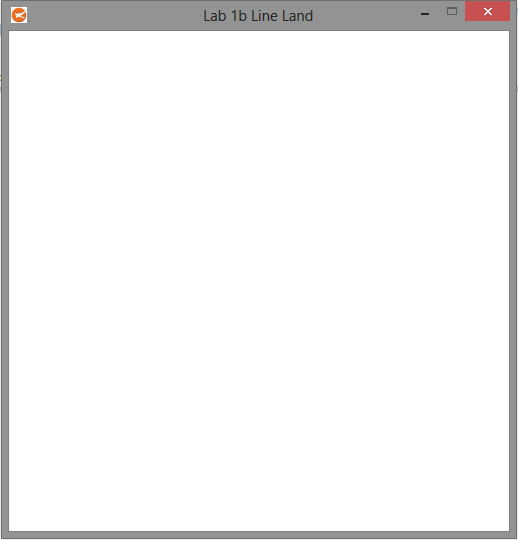
- Captain Jean-Luc Picard

# Required Materials

* Visual Studio 2015+.
* Windows 10 SDK.
* Provided D3D11 slides and fundamentals concepts taught in class.
* Book: “Practical Rendering & Computation with Direct3D11”.

# Setup

1. Create a new solution/project using visual studio 2015. This solution should be in a folder where you plan to keep all of your Direct3D hardware projects.
2. Our Direct3D applications will be designed to work on any OS after windows XP. (Vista & Later)
3. We want to make an empty win32 application using the wizard.
4. With your project created, add the main.cpp and XTime utility class to the project.
5. By default our project only supports 32bit platforms. We will want to add support for 64bit platforms as well. (If you already have ability this skip to step 10 )
6. Open the configuration manager panel. (found under the debug/release drop down)
7. Click the “Active solution platform” drop down and choose “new”.
8. By default it should say “x64” and copy setting from “win32”. This is what we want, click ok.
9. We now can choose between 32 & 64 bit builds, however by default there is no convenient drop-down. Lets remedy that:
   1. On the menubar go to View->Toolbars->Customize...
   2. Click on the "Commands" tab.
   3. Select the "Toolbar" radio button and find "Standard" in the drop down list.
   4. Click the "Add Command..." button.
   5. Select the "Build" category.
   6. Find and select the "Solution Platforms" command and click "OK".
   7. Place it next to the “debug/release” drop-down using "Move Up" and "Move Down".
   8. Enjoy not having to dig through solution properties to change the platform.
10. At this point you should be able to build your project in all four different configurations.
11. When you run your program you should see this:



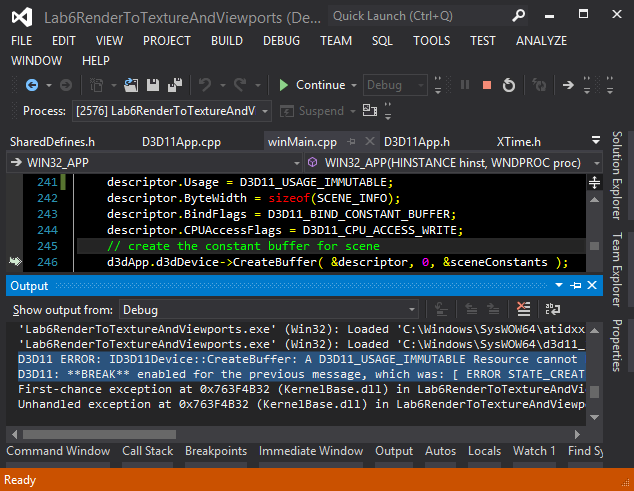
# MILESTONE 1: Clearing the Screen to Dark Blue.

1. Before we can do anything we will need to include the proper files and libraries to work with Direct3D11.
   1. Open your D3D11 Intro slides to the slide titled “Initializing D3D11” include and link to the two files listed. Using the Windows 10 SDK no special pathing is required to include and link these files.
   2. You will now need to include the DirectX math library; you will use this throughout future courses. Go to the last slide and follow the link to the math library docs. Find the “Getting Started” section. Make note of this page, this math library will be very helpful in Graphics II. For now just read the section “Basic Usage” to identify the proper header file. Once you have the proper header file included, go ahead and open it up. You will see a namespace containing all the classes and functions. Tell your main.cpp to “use” this namespace so you don’t have constantly re-type it.
2. In order to clear the screen we are going to need three D3D11 interfaces,(device, context, RTV) one DirectX Graphics Infrastructure interface,(swapchain) and finally a D3D11 structure.(viewport) The exact types for all of these can be found in the slides. Declare variables for all these types, classes starting with an “I” are COM objects and must be **Release**d during shutdown. If like me you feel releasing is a hassle and you would rather let a smart pointer deal with it, read this: <http://msdn.microsoft.com/en-us/library/ezzw7k98(v=vs.80).aspx>. Just be aware that you may need to look through the class to determine how to access the raw pointer.
3. The first three items we will need to make are the device, swapchain, and device context. There is a function listed in the slides that will actually create all three for us in one call. Find it.
   1. Before we call this function we must fill out a structure that describes the swapchain in detail. In the slides you will find this structure along with some commonly used parameters. Parameters not listed can be found in the MSDN, in case you are confused we are targeting a window(not fullscreen), and we don’t want any MSAA.(yet)
   2. You are now ready to call the function I mentioned. I want you to make a **hardware** device capable of at least **D3D10** capability. Use the **index in your book** to find the section on the “**d**evice”. I specifically want you to read the “creation” sub-section. After reading this section you should be able to call the function properly.

## PREPARING FOR DISASTER

I need to interrupt you for a second and tell you about one of the fantastic features of the D3D11 API!!! If you ask D3D11 politely it can and will carefully watch the values and data you try and feed it. If it detects you are using it improperly it has the ability to crash the program immediately! And here you thought crashes were a bad thing. =)

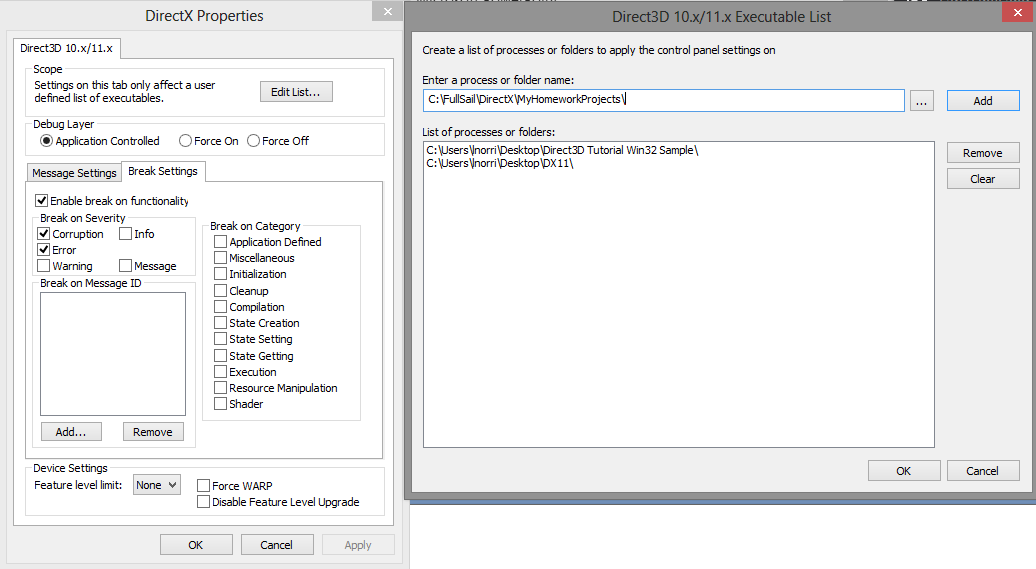
When this crash happens D3D11 will burp out to the Visual Studio “Output” window, as far as it can tell: what it thinks you’re doing wrong! This can be amazingly helpful! If you don’t turn this on, D3D11 just assumes you know what you are doing and doesn’t double check you. I don’t think I need to tell you that this might not be a good idea right now.



(^^^Example of D3D11 ERROR being shown in the Output Window^^^)

All right… so how do we turn this thing on??? The following steps will show you how!

* In the “DEBUG” menu in Visual Studio expand the “Graphics” section.
* Select “DirectX Control Panel”.
* Using a different window, navigate to your personal class folder and copy the full path.
* Make sure the above folder will contain ALL of your homework and exams for the entire class.
* Open the “edit list” window and paste the path in the top field and click Add.
* Under the “Direct3D 10/11” tab set the debug layer to “Application Controlled”.
* Switch to the “Break Settings” tab, select “enable break on functionality”.
* Enable **break**ing the application on corruption & errors.
* When you are done it should look something like this:



* The debug system is now ready to roll, but we still need to do one more thing.
* The debug layer is now “Application Controlled” so we must tell the application to use it.
* In the previous step in your homework you created the device.
* You may have noticed a “flag” parameter being passed to this function.
* I want you set this to “D3D11\_CREATE\_DEVICE\_DEBUG” **ONLY** if we are in **debug** mode.
* You can use the pre-processor to figure this out.(\_DEBUG)
* The next time your application crashes, look through the “Output” window… profit!

Ok… you just saved yourself many man years of pointless debugging! Back to the homework!

1. Our device, swapchain and context have now been generated. When you made the swapchain it also happened to generate the infamous “back-buffer” we discussed in lecture. The problem is though, that the backbuffer is buried somewhere in the swapchain and D3D11 is currently unaware of its existence.

The back-buffer itself is a rectangular array of colors. (AKA: a 2D Texture) To get access to it, you will need to get it from the swapchain. Open your slides and follow the “horse” on the slide which covers the swapchain. Read the article up to the “care and feeding” section. Come back when you are done. Ok good! Let’s use what you learned about to grab the back-buffer texture and create a render target view which points to that surface. Don’t set it just yet.

1. A viewport is used to define the area we will be drawing to on the render target. For now we will want to draw to the entire thing. Use the MSDN to look up the structure and fill it out appropriately. I want you to set the depth values to zero and one respectively.

(Pro-Tip: Instead of hard-coding the size, query it from the swapchain using “GetDesc”)

1. We are about start performing rendering operations. During shutdown it is important we free any outstanding COM object references that the device context may keep when you set resources to it. Use your book to find the sub-section “**A**pplication: shutdown sequence”. Read this section to find out how to free any bound resources from the context.

(If you used CComPtr<> ***some*** of this may be redundant)

1. With all of our resources and shutdown sequence now in place, we are ready to actually put them to use.
   1. The first thing we need to do is set our render target view to our context so we can actually draw to it. Look through the context member functions. The one you need should stand out at you. (hint: starts with “OM”)
   2. With our RTV set, we need to specify where on the target we wish to draw. Recall that the viewport contains this info. The context function required should be pretty easy to identify. (hint: starts with “RS”, I will explain these weird letters next time)
   3. Now we can finally clear the screen! Find the function that will do the job, and clear the screen to Dark blue. (hint: has the word “Clear” in the function)
2. Did you run it? Did it clear the screen like you asked???

If your screen is still white, it’s not because you got the color wrong. You must now transfer the back-buffer to the front buffer as mentioned in the companion slides. (Book: “**A**pplication: presenting to the window”) Once you get it working, you should see this:



This concludes milestone 1. (If you got here quickly proceed on!)

## IT’S NOT WORKING!!!

If yours is not working then you should definitely fix it before you continue to the next part. Hopefully yours is actually crashing, this is a good thing! Remember the debug setup step that you did earlier? Well if you have any luck the thing you did wrong is sitting staring at you in your “Output” window right after you crash in debug mode.(hint: look for “D3D11 ERROR:” in your message trail)

If all else fails, it’s time for some “specialist” assistance. However, try to make it a last resort for this particular assignment. She/he is just going to double check your work, did you do that already?

# MILESTONE 2a: Creating & Drawing a Circle.

1. To draw a circle we must first define what the circle consists of. Our circle will consist of lines connected by a set of 2D coordinates. Each coordinate will consist of a floating-point X and Y value. Make a structure to represent a single point, call it “SIMPLE\_VERTEX”. Inside the struct use an appropriate type from the math library you encountered earlier to describe the point. The “XMwhatever” type should contain only 2 values.
2. Ok! With our vertex defined we will need a way to store a collection of these values in VRAM so the GPU can get to them. The type of **I**nterface used by D3D11 to store various data on the GPU is mentioned in the slides when we talk about drawing a triangle. Go ahead and make one of these along with an unsigned integer used to count the number of vertices in our circle.

The last item required will be something else mentioned in lecture. This interface describes how a single vertex is “broken-down” into separate elements. Make one.

1. Now it is time to build our circle and upload it to VRAM! The circle must be made on the CPU RAM first before it can be copied to the GPU’s VRAM.
   1. First things first, let’s allocate enough vertices to make a circle. You can use the heap or stack, up to you. (just be sure to free any memory you allocate) To make creation of the circle simple lets allocate one vertex for every degree of the circle. With your vertices allocated create a loop to run though each of them. For each vertex use “sin” and “cosine” to calculate the current X & Y locations of the current “degree”.

(Don’t forget that math functions RARELY use degrees directly)

* 1. To copy our circle to VRAM, first we need to allocate space IN VRAM. For this we will need our buffer. To make a buffer we must first DESCribe it to D3D11. The slides show you how to describe a vertex buffer. (Use the MSDN for the remaining arguments!)
  2. With our buffer properly described, we need to tell it where the initial data used to populate its contents resides. This is an optional parameter when creating the buffer, however not specifying it will leave the buffer undefined.
  3. To create the buffer resource you must use the interface responsible for creating everything! Scan the member functions of this interface and find the create function we discussed in lecture.

1. So at this point we theoretically have a circle sitting somewhere in VRAM. The problem is; in order to draw anything with D3D11 you need something called a “shader”. In fact you need two of them at minimum. We don’t cover shaders till next time. So I have went ahead and provided you with the two you are going to need. That being said you are still going to have to compile them and load them into VRAM for use. Go ahead and declare two new D3D11 **I**nterfaces, one for a **vertex shader** and one for a **pixel shader**.(Find the exact types in the Book/MSDN)
2. Ok… pay close attention to this next part. Before shaders can be placed into VRAM they must be compiled. Any \*.hlsl (shader) files you add to your project will automatically get compiled into a \*.cso file. (compiled shader object) This is a new feature to VS 2012+. (An AWESOME feature!)

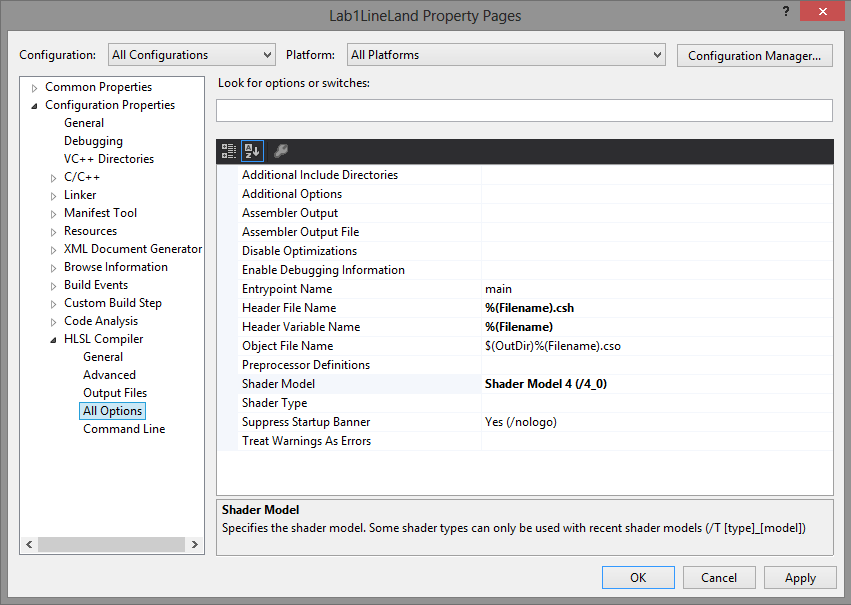
The thing is… there are many kinds of shaders. (Vertex, pixel, hull, etc…)To properly compile a shader you must configure it to be the proper type and utilize the appropriate shader model.

(We will get back to this in a minute)

A \*.cso file is essentially the compiled byte-code instructions of a shader (what the GPU understands) stored in a binary file. One option is to read the contents of this file into RAM using file IO. Then we would be able to copy this into VRAM using a function like “device:CreateShader”.

There is however… another option as well. The shader compiler built into visual studio also has the ability to take the compiled byte code and create a c++ header file which contains an array of bytes which contains the shader’s byte code instructions. You can then include this file in your project embedding the byte-code array within your executable file. This is the method I am going to show you.

* Open your solution explorer and add “Trivial\_VS.hlsl” to your project.
* Adding an \*.hlsl file to the project will enable the “HLSL Compiler” project settings.
* Right click on your project and select “properties”.
* Select the now available “HLSL Compiler” category.
* Now… before we alter settings we will want our changes to affect ALL builds.
* Make sure the 2 drop-downs say “All Configurations” and “All Platforms”.
* Edits we make here will also affect every hlsl file added to the project from here on out.
* First set the “shader model” to model **(/4\_0)** this is maximum shader model for D3D10.
* You will not require the capabilities of model 5 until day 8.
* Now, switch to the “Output Files” sub-category.
* Where it says “Header File Name” put:  ***%(Filename).csh*** (compiled shader header)
* What this does is generate a header file with the exact same name as your \*.hlsl file.
* Where it says “Header Variable Name” put:  ***%(Filename)***
* This will label the array containing the byte-code to also match the file name. (no ext)
* When you are done your overall settings should look like this: (Apply your changes)



* The last thing you need to do is to properly identify your shader files.
* Right-click on “Trivial\_VS.hlsl” and select “properties”.
* Once again, make sure the 2 drop-downs say “All Configurations” and “All Platforms”.
* Under “HLSL Compiler” make sure the type is set to “Vertex Shader”.
* Add “Trivial\_PS.hlsl” to the project and verify it has inherited the project settings.
* Mark it as a “Pixel Shader” and compile your project. (Don’t forget the drop-downs!)

1. At this point you should be generating two \*.csh files. Look inside them and verify that they both contain BYTE arrays matching their filename. This is the compiled byte-code ready for use.

#Include these files. We now have access to these arrays in our source code.

1. Now use the device to create the vertex and pixel shaders from the byte-code located in our new header files. The required device member functions should be self-explanatory, use the MSDN to be clear on what the parameters they take are.
2. Alright… whew… shaders are loaded! Can we draw now!? Nope sorry… not just yet. First we need to **glue** together the **data from** your **vertex buffer** to the **vertex shader input** **layout**.

Sounds familiar doesn’t it? There is a slide that talks about the input layout. Follow the horse on this slide and read the first two sections. (*How to draw a triangle: step 3.*)

* 1. Ok! Now that you have a pretty good idea what an input-layout object is, I want you to make me a descriptor array that defines ONE element bound to the “POSITION” Semantic. This element should use a **format** that defines TWO and ONLY TWO 32bit floating point values. The remaining arguments can match the values described in the article. (If you are curious, take a peek at the INPUT\_VERTEX structure at the top of your vertex shader. See anything familiar? Make a mental note of this for next time!)
  2. Anyways… let’s get back to it. You should now have everything you need to generate the input layout object. Once again you are **CREATING** something… do you recall which D3D11 interface does only creation? (Hint: The required function will access your vertex shader byte code for validation)

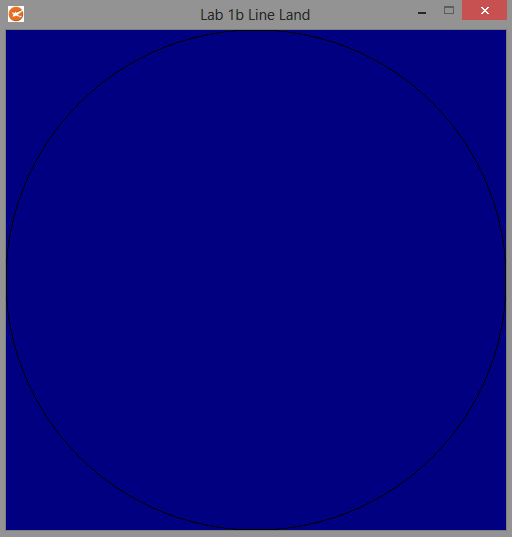
1. Can we please draw now??? Almost there! We are on the home stretch! D3D11 has no idea what it is you want to draw or how! Lets remedy that:
   1. Use the context to set your vertex buffer to the system so it knows **what** verts to use.

(Pay special attention to the “stride” parameter!)

* 1. If you are binding or drawing stuff, guess which interface you use? Now I want you to set the two shaders we created earlier so D3D11 knows **how** to draw something.
  2. Don’t forget the GLUE! That’s right; I want you to set the input layout next.
  3. What were we drawing again? Triangles? Points? Oh… that’s right, lines! To make a circle. You must tell D3D11 what kind of primitive you’re interested in drawing. Now would be a just swell time to **read** about **“P**rimitive topology” wouldn’t it?

(Truthfully, the time to think about this is **before** you make a shape not after hmm?)

1. I shouldn’t even have to say anything here, you are probably falling out your chair in anticipation to type the 4 magic letters that do this:



Well… that’s probably not as cool as you had hoped, but it’s something!

Does **your** circle have a tiny gap at the top? I know you could fix that…

In the next two sections we will focus on making it a bit prettier and a bit more active! If it’s not working, then it’s time to read the IT’S NOT WORKING!!! Section again.

The next part is pretty short! I wouldn’t stop unless you really need to.

# MILESTONE 2b: Make it Purdy!

1. Now we get to the good part, communicating with the shader via a constant buffer. So go ahead and make another buffer, and while you are up here toss in an instance of the XTime class. We will need that too at some point. (Might as well take a peek at it too while you can)
2. With our variables declared, we need to figure out what on earth this shader actually can do. Go ahead and take a peek inside the vertex shader source.
   1. See the structure sitting beneath this **section**? This is known as a “constant buffer”, we talked about it during lecture. It contains variables that are used to drive the shaders behavior. Notice the name of the structure, a shader is written in a language called HLSL. (High Level Shading Language) It exists only on the GPU, thus any buffers which affect it must also exist on the GPU. Though we cannot directly declare or modify GPU variables in C++, we CAN allocate data on the GPU and update it via the D3D11 API.
   2. Now that you have had a moment to study the constant buffer, I want you to make a **MIRROR** version of this structure in C++. Name your structure “SEND\_TO\_VRAM” and fill it with variables that match the type and sequence of the variables on the GPU side. To make things a bit more convenient try and use “XM” types from our math library.

(The “padding” variable’s only purpose is to maintain 16byte alignment, **TMI** right now)

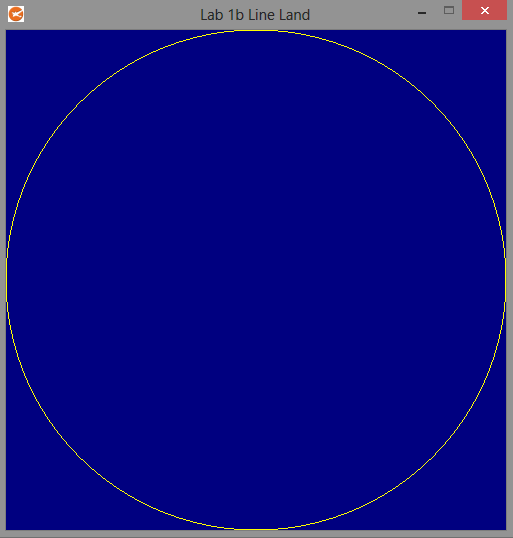
1. Now that we understand what we can send to the shader, let’s go ahead and allocate space in VRAM to hold this information like we discussed. The method is mostly the same as the vertex buffer we created earlier. However, we don’t need to fill in its initial contents right away as we will be doing that later.(The slides contain a wealth of information on constant buffers)
2. Great, so now we have a constant buffer reserved in VRAM. As you might imagine, allocated memory is not very useful until you actually fill it with something sensible. Let’s do that:
   1. Go ahead and make an instance of your new C++ structure, this is the CPU memory we will be copying into VRAM. (call it “toShader”)
   2. Now we need to initialize it so we don’t just copy garbage into VRAM. Set the offset value to the center of our screen (0, 0) and the color to full Yellow.

(Why is the origin not at the top left!? This will become clear next lecture)

1. Ok, so now we have both what we want to copy and where we want to copy it to. The slides explain two techniques for copying information into a constant buffer. Use the faster technique since this happens each and every frame. (Confused? Book: “**R**esources: mapping”)
2. Alright! Now the VRAM has been filled with the values we want! So… why isn’t the circle yellow? Well… just like with the vertex buffer, D3D11 is not aware of “which” constant buffer you want to use and where you want to place it. D3D11 actually supports up to 14 unique constant buffers to be available at any one time. We will need to connect your shiny new constant buffer to one of these 14 **“slots”**. The function used to do this is (you guessed it) part of the context. The name of the function you want should jump out at you. However, the first two letters you want might not be as clear. (Hint: what file is the actual constant buffer located in?)

Do you know which slot the constant buffer is expected to be found inside the shader? Look at the **“register”** section of the constant buffer and you will have your answer.

1. Umm… it’s still not working! Woops, sorry about that. Open your vertex shader and comment line 31 back in. This is what you should see:

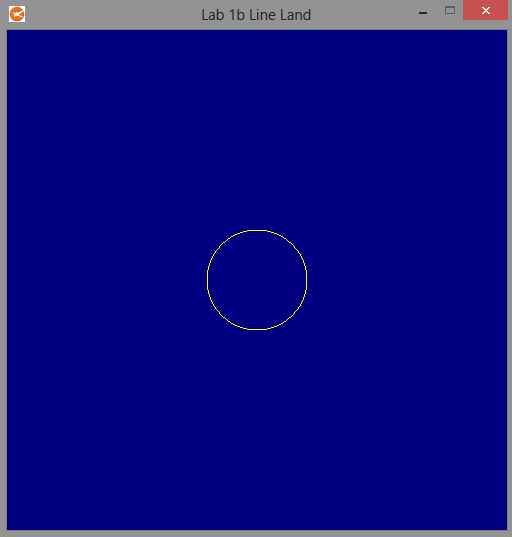


There, that’s a little bit better… this concludes Milestone 2.

The Final Milestone is a bit tricky but you will learn a lot about D3D11 & Hardware Shaders.

# Milestone 3a: Giving the Circle some Coffee.

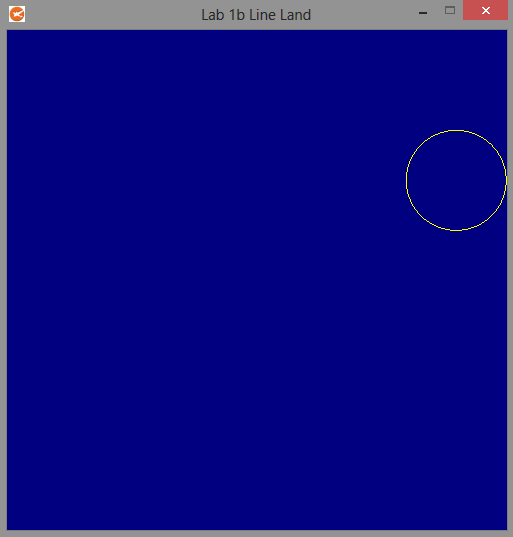
1. Our circle seems a bit boring, I think he’s consumed one too many radii. Let’s put him on a diet; create a loop which scales the vertex data down by a **factor** of 80%. You should get this:



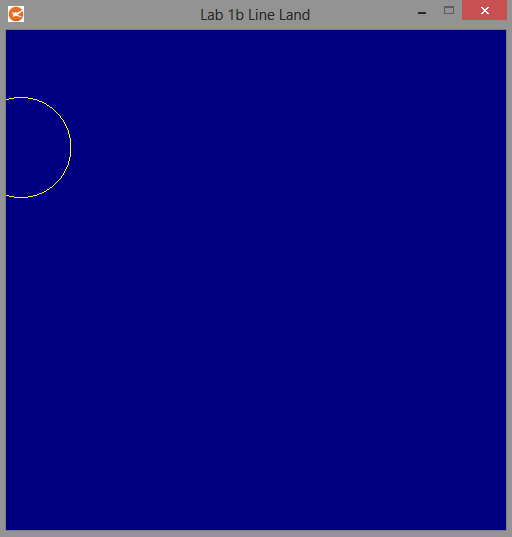
1. Well… now that he’s not so massive he should be able to move around. As you might imagine, drawing a tiny circle isn’t exactly the most taxing thing your video card has ever done. Thus, your frame rate is probably in the thousands right now. Asking the circle to move even a tiny bit (0.1) each frame will probably rocket him off to the moon. Instead let’s use some basic physics to move him a specific amount over a specific time interval. ((V=D/T) or Distance = Velocity \* Time)

Anyway, what I am trying to say here is we are going to need to know the amount of time passed in one frame. The XTime class has a function that you need to call every frame to mark down the change of time between intervals. Call this function here.

1. Create a 2D velocity vector of 1.0 on the X and half that much on the Y. Use the change in time to move the circle’s “**offset**” by this vector over multiple frames.
2. Why don’t you take a peek inside your vertex shader on line 28? Well there’s your problem! You should now see your circle take off to who knows where:



1. Ok great! Now that the circle is moving let’s make it so we can continue to see it by making it bounce off of the walls. The walls are as follows: Left Wall -1.0, Right Wall 1.0, Top Wall 1.0, and Bottom Wall -1.0. Use your 2D skills from earlier in the program to implement the bouncing behavior.



Only one more section to go!

# Milestone 3b: The Grid. A.K.A: What did I do to deserve this!?

1. Alright! Now it’s time to really challenge you! If you run the sample executable you will notice a black checker-board pattern overlaid across the blue background. We will be building this out of an array of triangles. Therefore we require yet another vertex buffer and count variable.
2. Remember when you built the circle in part 2? Unfortunately the grid will be significantly more difficult. You can do it though! Time to exercise those logic skills!
   1. The grid is built from a staggered set of 200 squares. There are 10 squares across and 20 squares going down. Each square is built from two connected triangles. You should know how many vertices it takes to make one triangle. Use this knowledge to allocate enough 2D vertex data to represent the entire grid.
   2. With your vertex data allocated it’s time to figure out the formula used to build the squares in a staggered formation. If you don’t get it right, don’t worry about it right now. Chances are you will be coming back to this step multiple times.

Here are some tips to make things a bit easier:

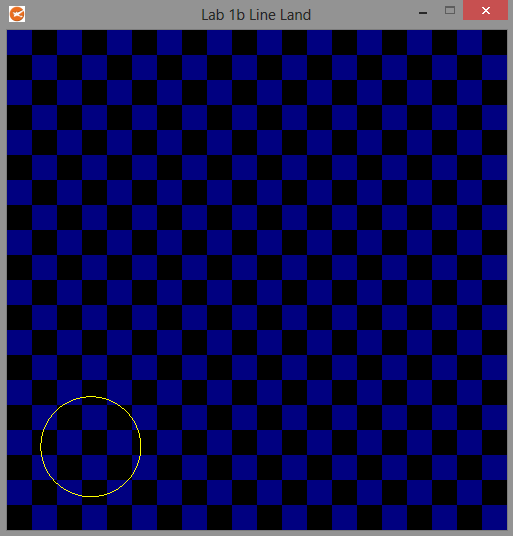
* Use a counter to track the current vertex you are on.
* Use a double for loop to go through the rows then columns.
* Each square is 0.1 across in both dimensions. (NDC)
* Start at the top-left wall and work your way across and down.
* Horizontal squares have a gap, vertically they don’t.
* Use modulus to determine if a row should be shifted by one.
* ALWAYS make sure your triangle’s point sequence is in a **clockwise** fashion!
* If you don’t do the step above you won’t be able to see your triangles!

1. So you got it right? Maybe? I guess we will find out…

While that certainly is a nice array you got there, do you know where it is? More importantly, do you know where it isn’t? Is it in VRAM? No? Can you fix that for us? (Hint: Part 2 step 3)

1. You know how our grid is yellow and bouncing around the screen? Oh wait… that’s not right, the grid supposed to be stationary and black. Hmm… well the shader currently has instructions to draw yellow bouncing stuff. Remember how we did that? Well, instead of asking the shader for yellow bouncing things let’s add a request for black stuff standing still. (Hint: Part 3 step 4)
2. Yup you guessed it! Before that request is actually going to do squat you need to copy it to VRAM. Use the exact same technique you used earlier.
3. Oh boy! Time to Draw!!! Hold the phone there speedy… aren’t you forgetting a thing or **six**? That’s right, D3D11 is confused! Are you drawing a circle? Is it made out of lines? How about you make yourself crystal, then we can move on. (Hint: You have had to do **ALL** of this once before)
4. If you made zero mistakes or are some kind of sorcerer you should see the following:

(If you are a mere mortal like the rest of us, you should probably go back and review steps 2 & 6)



Do you have this? Awesome! You are done!

If you feel pretty confident with this stuff, read the extra credit section in the Syllabus.

# Conclusion

Congratulations, you have reached the end of the HW Project! Pat yourself on the back!

Did you notice something? How many lines of code did you copy from this document? None!? How can that be? As you have discovered, information for how to operate this API exists in abundance. The tough part will be honing your ability to find and process this data into usable code. This, alongside the theory you learn in lecture and the reading will prepare you for later courses. Good Luck!

If you are eager to start messing around in 3D using hardware, it is no different than what you did in Lab 4 for software. Just build your World, View & Projections matrices and transmit them to the Shaders via constant buffers. One Critical Detail: HLSL assumes all matrix data is **COLUMN MAJOR** by default. Do not forget to **TRASNPOSE** your matrices before submitting them to HLSL!!!