VideoGraphics: AnaGame’s 3D Library

The VideoGraphics library is AnaGame’s 3D support library and (currently) the second lowest library in the AnaGame stack. It’s job is to abstract away 3D API specific functionality and provide a platform for 3D programming, regardless of the underlying 3D API used, whether it is Direct3D or OpenGL (or others such as Vulcan or Mantle).

Currently, the Windows version uses Direct3D and supports HLSL shaders. However, Direct3D is only available on Windows and thus HLSL files would not work if AnaGame is ever ported beyond Windows. Unix-versions of AnaGame will most likely use OpenGL and GLSL as it’s shading language.

In fact, If AnaGame is to support WebGL, then it would have to support GLSL in some capacity, even on the Windows version. However, this qualifies as an *eventual* to-do (a problem to address eventually, but not in the immediate future).

Before discussing how VideoGraphics works, let’s look into how to set it up…

# Set-up

Obviously, this is a rehash of what might be found in a couple README’s of the repository but I feel it is necessary to mention it here. If you have already followed these README’s and gotten VideoGraphics to successfully build, you can safely skip this section and go to page 6.

Note: Build the TrecLib Library first (and the DirectXTex library from Microsoft in the following subsection).

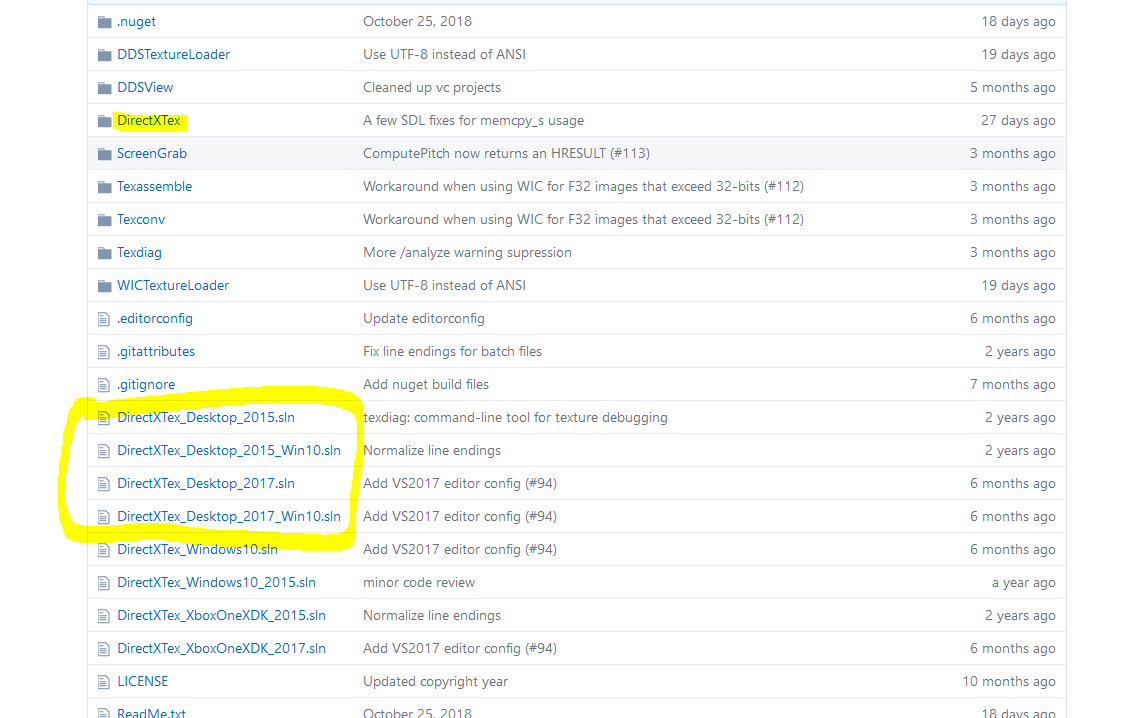
## 3rd Party Library (Textures)

When Microsoft created the Windows 8 Software Development Kit, they integrated the DirectX Kit in with it. However, this kit has gaps in the library that prevents it from fully supporting textures, requiring a third-party library to provide this functionality.

The library can be found here: <https://github.com/Microsoft/DirectXTex>

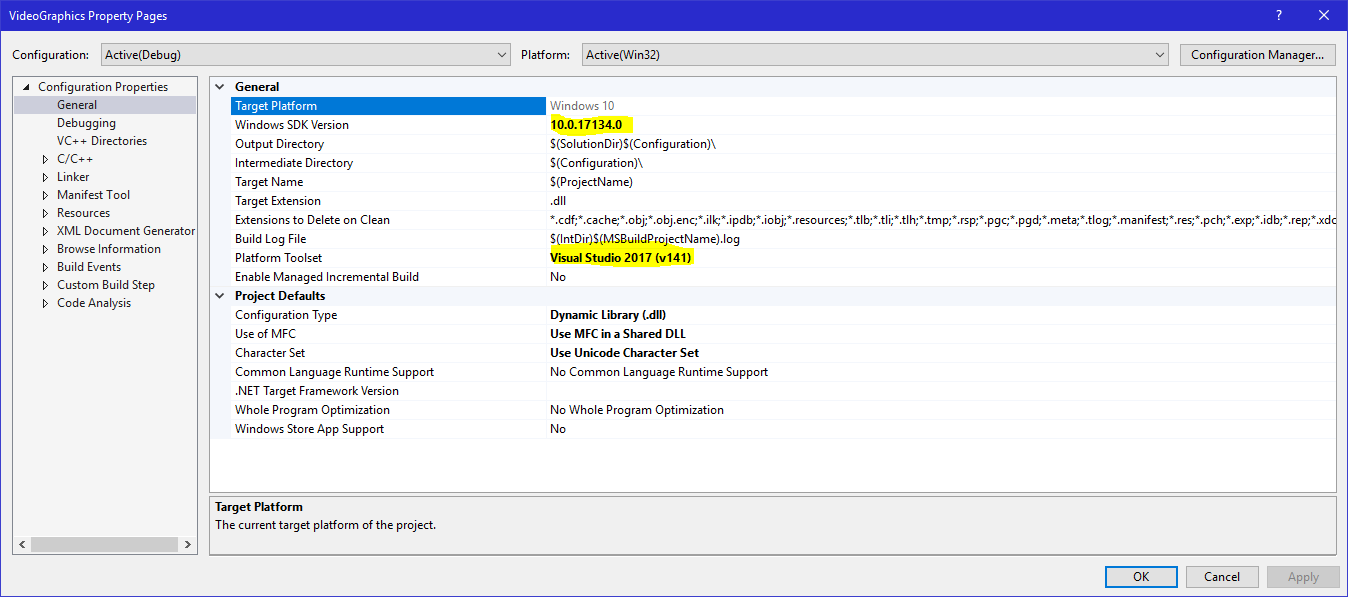
If you look at the image on the following page, the directXTex Folder is the one we’re interested in. It contains the header files for Texture functionality. I also circled the Solutions used to compile these libraries. Choose **1** solution depending on your IDE/Windows version.

* If you’re using Visual Studio 2017, use a 2017 Solution
* If you’re using Visual Studio 2015, use a 2015 Solution
* If you’re using Visual Studio 2013, you’ll need to clone an earlier version of the DirectXTex library with 2013 supported.
  + The latest version with 2013 I found is located here: <https://github.com/Microsoft/DirectXTex/tree/99cf8e712db23828e7b13614e132b4e1afe7d8f6>
* If you’re using Windows 10, then the *Win10* version should be safe. Otherwise, pick the other version.



Build the solution you pick. After that, it should produce a series of libraries, one of which you’ll need for the next step.

## Project Settings

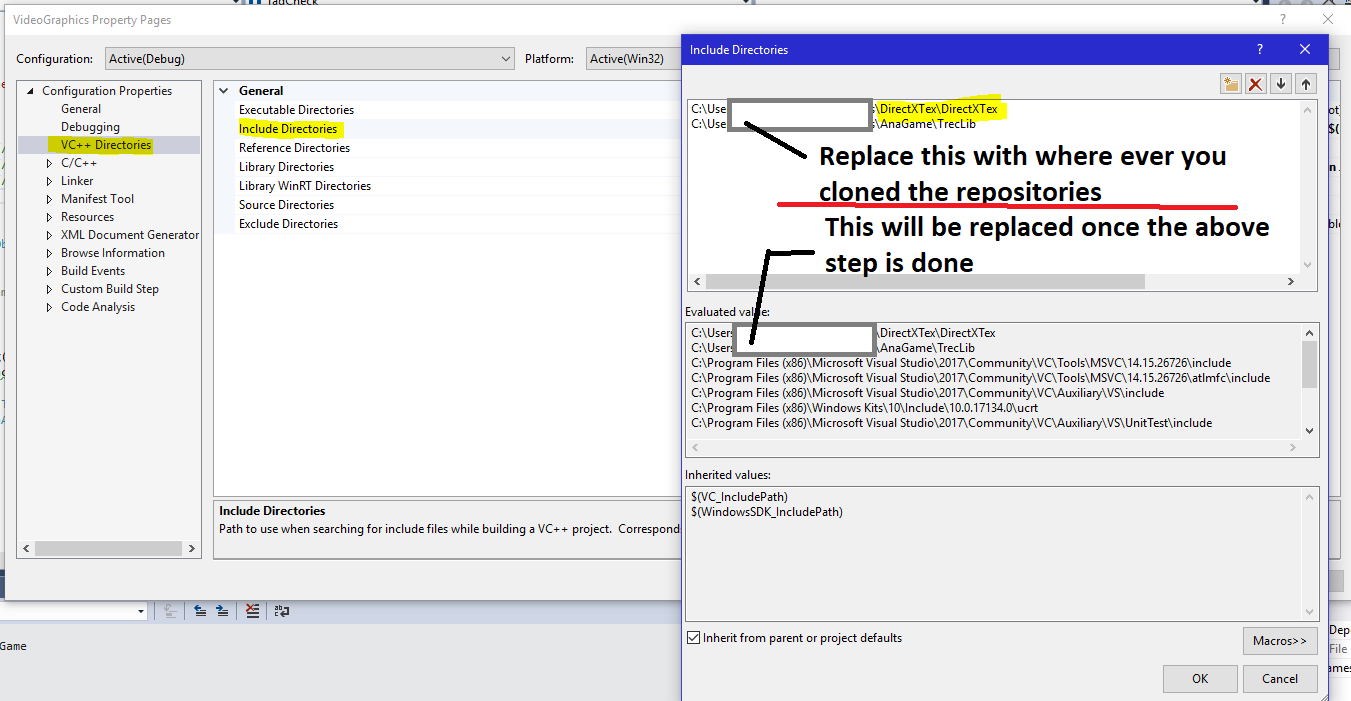


You should make sure that the settings are properly configured. I have highlighted the two most likely to cause problems. Simply set them to a value supported by your Visual Studio installation (and Operating System).

When I first cloned AnaGame to a second machine, the SDK Version was set to 8.1, which was not on my machine.

## Include Settings

Go to “VC++ Directories” 🡪 “Include Directories”



X – out the fields in the white box in the focused dialog and replace them with where the folders are on *your* machine.

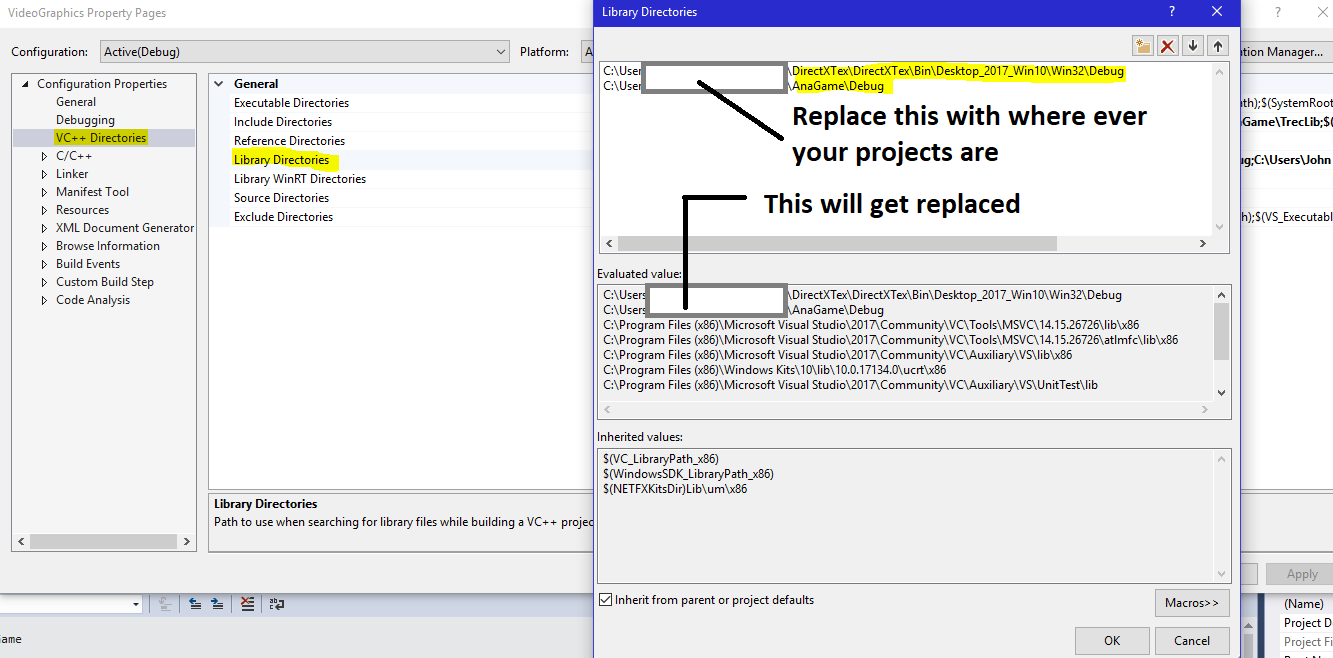
## Library Settings

Go to “VC++ Directories” 🡪 “Library Directories”

The image below might be hard to read. The Directores to be adjusted are as follows:

* DirectXText/DirectXTex/Bin/Desktop\_2017\_Win10/Win32/Debug
* AnaGame/Debug

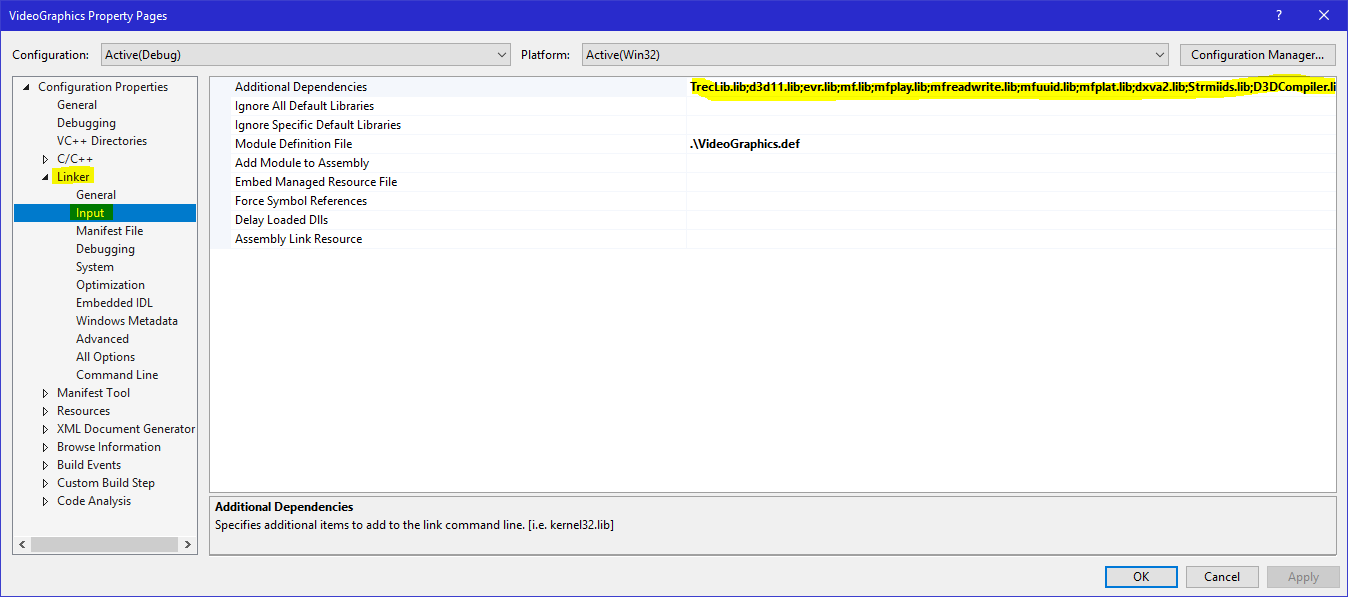
Note, I used the Desktop 2017 version for Windows 10. The exact path for the first library depends on which solution you used to build it.



## Lib Files

Last but not least are making sure the Lib Files are included. This is one step you likely won’t have to change but it is good to check on it just in case.

“Linker” 🡪 “Input” 🡪 “Additional Dependencies”



Although each library should be listed already, here is a list of what VideoGraphics uses:

* TrecLib.lib
* d3d11.lib
* evr.lib
* mf.lib
* mfplay.lib
* mfreadwrite.lib
* mfuuid.lib
* mfplat.lib
* dxva2.lib
* Strmiids.lib
* D3DCompiler.lib
* DirectXTex.lib

# ArenaEngine: the 3D Environment manager

->TML

-|Type:ShaderSource

-|Version:0.0.1

-/

-|InputSlot:0

-|BufferPurpose:Position

-|BufferWidth:4

-|BufferSize:3

->Buff

-|PixelFile:sColor.fx

-|PixelFunction:PS\_Main

-|VertexFile:sColor.fx

-|VertexFunction:VS\_Main

-|ConstantSize:64

-|ConstantShaderPhase:Vertex

-|ConstantSlot:0

-|ConstantPurpose:Model

->Const

-|ConstantSize:64

-|ConstantShaderPhase:Vertex

-|ConstantSlot:1

-|ConstantPurpose:View

->Const

-|ConstantSize:64

-|ConstantShaderPhase:Vertex

-|ConstantSlot:2

-|ConstantPurpose:Camera

->Const

-|ConstantSize:16

-|ConstantShaderPhase:Pixel

-|ConstantSlot:3

-|ConstantPurpose:Color

->Const

The ArenaEngine is the main class used with AnaGame 3D development. It’s job is to manage shaders, hold a collection of models, hold the resources needed to draw to a specific window, and make calls to the underlying 3D API.

## Shaders

In the context of AnaGame, there are two major types of shaders: AnaGame shaders and External shaders. AnaGame shaders are statically managed, meaning there is one collection for *all* instances of ArenaEngine. These are the shaders AnaGame provides internally and will compile when the first ArenaEngine is made. It will also clean up these shaders once the last ArenaEngine is destroyed.

While ideally, the AnaGame shaders should be sufficient (once AnaGame is more advanced), there may come a few cases where they don’t produce the desired effect. For cases like these, AnaGame allows for External shaders to be provided. External shaders are managed by each individual ArenaEngine.

Because AnaGame was designed with a wide-variety of shaders in mind, it does not limit vertex input to a specific type (in contrast to most DirectX (and possibly other 3D API) tutorials). Because shaders could have a wide variety of features and layouts, AnaGame needs a mechanism to manage this variety and interact with them correctly. Therefore, it uses TML.

The code snippet to the left is an example of the Shader TML language. There are two major objects in the current version of the standard: *Buff* and *Const*. The *Buff* object represents a section of the vertex layout. The *const* object represents constant shaders

### Buff: Vertex Input

If you’re not familiar with how shaders work, here is a brief summary:

Shaders are small programs that run on the GPU. They form a pipeline that takes in vertices, processes them, and renders them on screen. There are two **mandatory**shaders in the pipeline: Vertex and the Pixel/Fragment shader.

The Vertex shader is the shader that runs first and it’s job is to perform calculations on the vertices one at a time. The verticies can include an actual 3D vertex, 4D Vertex (where the fourth number should just be 1, a texture coordinate (coordinate on a given image), or some color.

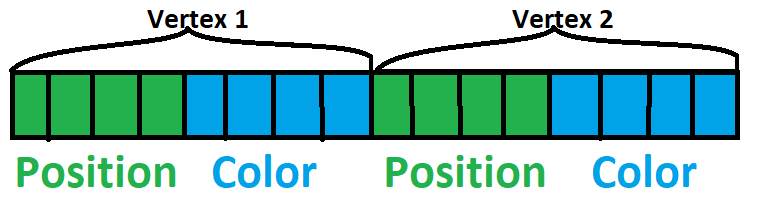
This is where the vertex input “buff” “object” comes in regarding the TML file. We see that there are four attributes associated with the buff object: InputSlot, BufferPurpose, BufferWidth, and BufferSize.

InputSlot: “Channel” to send verticies through, safe to stick with 0

BufferPurpose: What does *this* section of the vertex represent? Possibilities include *Position*, *Color, Normal, Texture*, amongst others.

BufferWidth: the number of bytes per variable in the Vertex Buffer

BufferSize: the number of variables in the section of the buffer



Looking at this sample input, we have 2 Vertices being inputed into a hypothetical Vertex Shader. Each vertex has 2 sections: a Position section and a Color section. Each section has four cells (the buffer size). If we assume that each cell represents a 32-bit floating point value (as is common), then the Buffer width would be 4 (as in 4 bytes). So the TML for this shader might look something like this 🡪

-|InputSlot:0

-|BufferPurpose: Position

-|BufferWidth:4 # Number of bytes per cell

-|BufferSize: 4 # Number of cells per section

->Buff # Complete and submit the section of the vertex

-|InputSlot:0

-|BufferPurpose: Color

-|BufferWidth:4 # Number of bytes per cell

-|BufferSize: 4 # Number of cells per section

->Buff # Complete and submit the section of the vertex

### Const: Persistent data

The Vertex Data changes for every vertex sent to the shaders/GPU. But what if there is some data we wish to be constant across multiple vertices. This is where *Constant Buffers* in the shader and the *const* “object” come into play in AnaGame.

There is a *required purpose* in AnaGame for these constant buffers. That is to represent the Model-View-Camera set-up common in 3D environments. The *Model* represents the direction and location of a model (set of vertices). The *View* represents the location/direction of a *camera[[1]](#footnote-1)*. The *Camera* represents the perception of the camera.

There are two ways to represent this set up in shaders provided to AnaGame:

1. an *MVP* constant buffer (in which case the three matrices in the Model-View-Camera set up are multiplied by the software (C++))
2. three constant buffers marked as *Model*, *View,* and *Camera* – in which case the matrices are multiplied together by the shaders per vertex

The following table showcases the two mechanisms in TML:

|  |  |
| --- | --- |
|  |  |
| 3 Constant Buffers | 1 MVP Constant Buffer |
| -|ConstantSize:64  -|ConstantShaderPhase:Vertex  -|ConstantSlot:0  -|ConstantPurpose:Model  ->Const  -|ConstantSize:64  -|ConstantShaderPhase:Vertex  -|ConstantSlot:1  -|ConstantPurpose:View  ->Const  -|ConstantSize:64  -|ConstantShaderPhase:Vertex  -|ConstantSlot:2  -|ConstantPurpose:Camera  ->Const | -|ConstantSize:64  -|ConstantShaderPhase:Vertex  -|ConstantSlot:0  -|ConstantPurpose:MVP  ->Const |

ConstantSize: the number of bytes taken up by the buffer

ConstantShaderPhase: Which shader uses it? Vertex shader? Pixel/Fragment shader?

ConstantSlot: Where in the GPU’s memory slots does this go (each constant buffer MUST have it’s own slot 🡪 they cannot be shared)

ConstantPurpose: What does this Cosntant buffer do, used by AnaGame to know which constant buffer to update based off of the information it’s provided during runtime.

Shaders are allowed to have other buffers available, provided AnaGame knows about them, such as a Color buffer.

### Files: the Actual shaders themselves

So far, we have looked at TML files that outline the data that shaders work with. But these TML files are a Blueprint AnaGame uses to interact with the shaders – they are not the shaders themselves. If you look back at the long example above, four attributes were marked in red.

-|BufferSize:3

->Buff

-|PixelFile:sColor.fx

-|PixelFunction:PS\_Main

-|VertexFile:sColor.fx

-|VertexFunction:VS\_Main

-|ConstantSize:64

-|ConstantShaderPhase:Vertex

These four attributes tell AnaGame what files the shaders are actually in and what function name to compile. Without these files, AnaGame cannot build a shader program.

The examples seen here incorporate a Pixel shader and a Vertex shader. However, other shaders include “Domain”, “Compute”, “Hull”, and “Geometry” with “Function” and “File” appended after each of the proceeding.

## Constructing Shaders

The process of Constructing a shader is a long one and must be broken up into sub sections.

Phase 1: Setting up a Parser

1. The TML file to use (with shader files stored relevant to it) is determined
2. A File is set up opened to that TML file (currently, the default shaders use the CFile/CArchive – which should be replaced by the TFile)
3. The TML Reader and Shader Parser are constructed.
4. If an AnaGame shader is being built, the *SetDefaultShader* method of the *ShaderParser* is called. Otherwise, the Parser assumes an external shader is being constructed.
5. The Reader’s *read* method is called

Phase 2: The Parser Builds the Shader Model used by the Engine, sometimes by making calls to the Engine itself

* If the Parser gets a constant buffer attribute, it updates its own *local ConstantBufferDescription*.
* If the Parser gets a “Const” Object:
  + Parser calls ArenaEngine’s *GetContantBuffer* method with the size of the constant buffer and a Reference to the buffer pointer.
  + If the Buffer is generated, then the Engine’s *SetNewConstantBuffer* is called on the correct shader held by the engine.
  + A *ConstantBufferMark* – containing both a buffer reference and a mark detailing the buffer – is set and pushed back onto the constant buffer array.
  + Depending on the purpose of the constant buffer, indexes for crucial constant buffers are set accordingly
* If the Parser gets an input attribute,
  + The Parser calls either *SetDataCount*, *SetDataWidth*, *SetInputSlot*, or *SetBufferPourpose*, depending on the specific attribute.
  + The USHORT *desc* attribute is updated
* If the Parser gets a “Buff” object, it will add the *desc* attribute to the TDataArray for use in the Input Layout assembly.
* If the Parser Gets a File/Function attribute
  + The File version of each shader needs to come before the Function attribute
  + If the Attribute is for the Pixel or Vertex Shader, the Parser’s *BasicShaderDetails* are updated
  + If the Shader is a non-pixel/vertex shader, then the Parser’s *phase* is set and the Parser’s *AddShaderToProgram* method is called – which in turn calls the Engine’s Add\_\_\_Shader[[2]](#footnote-2) method (which attempts to compile the shader and add it to its shader object
* After each attribute is called, the Parser calls it’s *SetBasicShader* method, which is covered in Phase 2.5.

Phase 2.5: Setting up the basic shader (called 2.5 because it’s large enough to be grouped in it’s own phase yet could complete before phase 2 does)

Phase 2.5 commences under the following conditions:

1. The *BasicShaderDetails* is all set (all of its bools are true)
2. Phase 2.5 has not already been run for the current shader object.

The steps:

1. *SetBasicShader* is called, which will prepare the file and function parameters for both the Vertex and Pixel shader.
2. It calls the Engine’s *SetNewBasicShader* method (the version that takes two files and functions)
3. The Engine set’s a new Shader Object with it’s default attributes.
4. The Engine then calls *SetNewBasicShader* for just the Vertex Shader
   1. The second Method attempts to compile the raw shader from the file/function provided.
   2. The Vertex Shader is created and assigned to the Shader object.
   3. *getInputDescription* is called, whose job it is to convert an AnaGame input description into a DirectX input description. The procedure is pretty straight forward once you understand both the D3D11\_INPUT\_ELEMENT\_DESC structure and how AnaGame holds it’s data in a USHORT (see Appendix C for details)
   4. the ID3D11Device’s *CreateInputLayout* is called and its results are assigned to the Shader Object
5. The engine calls the same method as in step for, but for the Pixel shader
   1. See 4a
   2. The Pixel Shader is created and assigned to the Shader Object
   3. Because the Pixel Shader does not need Input layout information, the method simply returns.

## Rules for setting a Successful Shader with TML

1. For both the Input layout and Constant Buffer “*Objects*”, set their attributes before the object itself.
2. Set the Input Layout *Buff* objects first. When the Engine sets up a basic shader, it expects to have an input layout available to work with.
3. After the Buff objects are set, set the attributes for the *Pixel* and *Vertex* shaders next.
   1. AnaGame needs both shaders to work properly.
   2. If you set an alternative shader or constant buffer before setting the Pixel/Vertex shaders, AnaGame will reset the ShaderObject and the constant buffer and other shaders will be lost.
4. For each shader phase, Set the File attribute before each Function. The Function attribute is when AnaGame will attempt to compile the shader.
5. Group the shader attributes by shader phase. (if you set the Hull file and then the Compute function, for example, AnaGame will get confused over which phase to use and Direct3D will fail to compile it)

## Rules for writing an AnaGame compatible shader

1. Make sure that your shader program contains both a Vertex and Pixel shader
2. Make sure you have either 1 64-byte constant buffer set to *MVP* or 3 64-byte constant buffers set to *Model, View*, and *Camera*.
3. Use floating-point values for input data.

# ArenaModel

# To-Do’s

1. Rewrite the Texture Shader’s to use input layout of the single color shader
2. Test and improve AnaGame’s use of textures
3. Slowly add shader’s that utilize other features of 3D Cards, such as lighting
4. Build pipeline to support video playback, liquids, etc.
5. Save model’s to disk
6. Debug, Debug, Debug (and patch up any security vulnerabilities that might exist

## Eventual To-Do’s:

1. Collision Detection
2. Implement a loop that allows gameplay
3. Find a way to either support OpenGL or support GLSL with DirectX and emulate OpenGL for WebGL
4. Develop scripting mechanism to manipulate models

# Appendices

## Appendix A: Shader Parser’s BasicShaderDetails

typedef struct BasicShaderDetails

{

TrecPointer<TString> vertexFile;

bool vertexFileSet;

TrecPointer<TString> pixelFile;

bool pixelFileSet;

TrecPointer<TString> vertexFunction;

bool vertexFunctionSet;

TrecPointer<TString> pixelFunction;

bool pixelFunctionSet;

}BasicShaderDetails;

This structure is used by the parser to track whether the File and Function strings are set for both the Vertex and Pixel shader. These two shaders are the most crucial for 3D rendering and thus AnaGame will not build your shader object until these two shaders are ready.

## Appendix B: Shader Parser’s ConstantBufferDescription

This is a structure used and set by the Parser to manage information about a shader’s constant buffer. This structure’s attributes are set when an attribute is set and then pushed onto an array once the object is declared (hence why you set the attributes **before** the object itself.

/\*

\* struct ConstantBufferDescription

\* Purpose: Provides information regarding a shaders constant buffers

\*/

typedef struct ConstantBufferDescription

{

ShaderPhase sp; // Shader phase the Constant buffer belongs to

int bufferSlot; // the slot in the GPU this buffer is located

bool ModelBuffer; // Whether this buffer is for Model data

int size; // Size in bytes of the constant buffer

unsigned char purpose; // What this buffer is used for (used by AnaGame)

}ConstantBufferDescription;

For *bufferSlot*, make sure this attribute is different for each constant buffer.

*ModelBuffer* may be a redundant attribute, as the purpose could be used for whether it is a model buffer or something else.

*Purpose* is coded according to the following:

1. (“Color”) the buffer is used for a color
2. (“Camera”) The buffer is a 64-byte projection matrix (GPU does MVP calculation)
3. (“MVP”) The buffer is a 64-byte MVP matrix whose contents are calculated and updated from the CPU
4. (“View”) The Buffer is a 64-byte matrix specifying the location and direction of the camera
5. (“Model”) The Buffer is a 64-byte matrix specifying the location and direction of the model being drawn.

## Appendix C: the USHORT specifying the Input Layout

Although Microsoft has dedicated the D3D11\_INPUT\_ELEMENT\_DESC structure for managing layout information, AnaGame has dedicated a USHORT to hold the same information.

0b0000111100001111

Input Purpose: What the input value is used for (such as Vertex, Color, Texture, or something else

|  |  |  |
| --- | --- | --- |
| Value | AnaGame Attribute | Microsoft String |
| 0 | Color | COLOR |
| 1 | Normal | NORMAL |
| 2 | Texture | TEXTCOORD |
| 3 | Position | POSITION |
| 4 | Binormal | BINORMAL |
| 5 | Blendweight | BLENDWEIGHT |
| 6 | Blendindex | BLENDINDEX |
| 7 | T Position | POSITIONT |
| 8 | Tangent | TANGENT |
| 9 | Fog | FOG |
| 10 | Tessfactor | TESSFACTOR |

Note: I personally am only familiar with the first four (and even Normal, I’m sketchy on). There was even a previous format that only supported the first four values, though this has since been dropped.

Data Width: The Size of each field (or cell in the earlier image) in the input section. If using a regular 4 byte integer or float, use 4. For a Double or a Long Long, use 8. The Current maximum is 15 (though most systems won’t need to use more)

Input Slot: There are actually only two logical values Direct3D accepts. I recommend keeping it 0 unless you know what you’re doing. I’m actually not sure what the other value will do.

Data Count: The number of fields in the section. Typically, this number is usually three or 4. Again, the max is 15

## Appendix D: Engine’s ConstantBufferMark

Holds a shader object’s constant buffer as well as a marker signifying what that buffer does.

/\*

\* struct ConstantBufferMark

\* Holds a shader's constant buffer as well as a label signaling what that buffer is

\*/

typedef struct ConstantBufferMark

{

unsigned char label;

TrecComPointer<ID3D11Buffer> buff;

}ConstantBufferMark;

The buff attribute is self-explanatory, so the label should be looked at: 0b11001100

Purpose of Constant Buffer: What the constant buffer is used for - List available at the end of Appendix B

Is Buffer Model Buffer: Boolean

Slot: Which GPU register points to the buffer.

## Appendix E: Engine’s ShaderProgram (The Shader Object)

This is the structure that holds all the information needed to manage an AnaGame compliant shader, holding all the vertex, pixel, etc. shaders a given shader uses. It also holds input layout information in terms of what the Vertex shader expects.

There is also a list of constant buffers as well as a series of indexes pointing to the proper constant buffer for a given piece of data (which are initially set to -1 until properly assigned.

/\*

\* struct ShaderProgram

\* Holds all of the resources that constitute a shader as well as indexes to imprtant

\* resources within the shader

\*/

typedef struct ShaderProgram

{

TrecComPointer<ID3D11VertexShader> vs;

TrecComPointer<ID3D11ComputeShader> cs;

TrecComPointer<ID3D11DomainShader> ds;

TrecComPointer<ID3D11GeometryShader> gs;

TrecComPointer<ID3D11HullShader> hs;

TrecComPointer<ID3D11PixelShader> ps;

D3D11\_INPUT\_ELEMENT\_DESC\* elements;

unsigned char elementCount;

int bufferSize;

TrecComPointer<ID3D11InputLayout> layout;

HRESULT layoutError;

TDataArray<ConstantBufferMark> constantBuffers;

signed char cameraLoc;

signed char colorLoc;

signed char modelLoc;

signed char viewLoc;

signed char TextureCount;

bool mvp\_cpu;

}ShaderProgram;

1. Note the Camera is represented by the TArena control in the *Anaface* library. [↑](#footnote-ref-1)
2. The underscore can represent *Domain, Compute, Hull* or *Geometry*, depending on the shader phase in question [↑](#footnote-ref-2)