HLSL In-Depth

Topics

- Data types & operators
- Registers & semantics
- Flow control
- Intrinsic functions
- Custom functions
- Examples

Data Types & Operators

Major Data Types in HLSL

- **Scalars**: One-component
- **Vectors**: 1 to 4-component
- **Matrices**: 1 to 16-component (1x1 up to 4x4)
- Other types:
  - Textures
  - Samplers
  - User-defined Structs

Major Scalar Types

- **bool**: true/false
- **int**: 32-bit signed integer
- **uint**: 32-bit unsigned integer
- **float**: 32-bit floating point value

- Other advanced types exist
  - Above types are good enough in most cases

Scalar Types - MSDN
Scalar Literal Grammer - MSDN

Vector Types

- Contains between 1 and 4 scalar components
  - Yes, HLSL supports 1-component vectors
  - Yes, they're basically scalars
- All components must be of the same type
- Can be declared in multiple ways
  - Built-in type declaration
  - Template (generic) declaration

Vector Type Declaration

- Built-in type declaration:
  - `[scalar type][number of components] varName;`

  - `float3  direction;`
  - `int4    otherData;`

- Template (generic) declaration:
  - `vector<type, number of components> varName;`

  - `vector<float, 3>  direction;`
  - `vector<int,   4>  otherData;`

Vector Initialization Examples

```
// These are all valid

float4 color = float4(1.0f, 0.5f, 0.0f, 1.0f);

float3 direction = { 1.0f, 0.0f, 0.0f };

int1 dumbVector = 6;

vector<int, 2> twoInts = { 5, 10 };
```

Vector Components

- A vector's components may be accessed with:
  - r, g, b, a
  - x, y, z, w

- Accessing data in colors (or ANY vector):

```
float4 color      = float4(1,0,0,1);
float redAmount   = color.r;   // or color.x
float greenAmount = color.y;   // or color.g
```

Swizzling

- Syntactical shortcut for rearranging vector components

```
float4 color     = float4(1.0f, 0.5f, 0.1f, 0.0f);
float4 foSwizzle = color.xxyx;  // 1, 1, 0.5, 1
float4 worksFine = color.agbb;  // 0, 0.5, 0.1, 0.1
float4 error     = color.xbya;  // Can't mix notations
```

Swizzling

- If fewer components than necessary are specified, the last one is repeated

- When a float4 is expected:
  - .xy really means .xyyy
  - .z really means .zzzz
  - .xxx really means .xxxx

- Code examples:

```
float4 color = float4(1, 0.5f, 0.25f, 0);
float4 allZ  = color.z;  // 0.25, 0.25, 0.25, 0.25
float4 someY = color.xy; //    1,  0.5,  0.5,  0.5
```

Matrix Types

- Contain between 1 and 16 scalar components
  - Yes, HLSL supports 1x1 matrices
  - Yes, they're basically scalars
- All components must be of the same type
- Can be declared in multiple ways
  - Built-in type declaration
  - Template (generic) declaration

Matrix Type Declaration

- Built-in type declaration:
  - `[scalar type][rows]x[columns] varName;`
  - `float4x4  worldMatrix;`
  - `int3x1    otherData;`
- Template (generic) declaration:
  - `matrix<type, rows, columns> varName;`
  - `matrix<float, 4, 4>  worldMatrix;`
  - `matrix<int,   3, 1>  otherData;`
  - `matrix viewMatrix; // Shortcut for float4x4`

Matrix Initialization Examples

```
// These are all valid

float2x2 mat = float2x2(1.0f, 0.0f, 1.0f, 0.0f);

float2x2 mat2 = { 1.0f, 0.5f, 1.0f, 2.0f };

matrix<float, 2, 2> mat3 = { 1.0f, 0.0f,
                             0.0f, 1.0f };
```

Matrix Casting

- Sometimes you want a portion of a matrix
  - You can cast to the portion you want

```
// Declare a 4x4 matrix (and probably
// give it some data)
float4x4 world;

// Get the top left 3x3 portion
float3x3 topLeftPart = (float3x3)world;
```

User-Defined Structures

- Made up of one or more member variables
  - Similar to structs in other languages
- Useful for passing multiple pieces of data between shader stages
  - Sending data from Vertex Shader to Pixel Shader
  - Returning multiple colors from Pixel Shader
- Useful for defining shader input data too

Struct Examples

```
struct DirectionalLight
{
    float4 Color;
    float3 Direction;
};

struct Material
{
    float4 Color;
    float2 TextureOffset;
    float2 TextureScale;
};
```

HLSL Operators

- Basic Math: +, -, *, /, %
- Assignment: =, +=, -=, *=, /=, %=
- Boolean: &&, ||, ?:
- Comparison: <, >, ==, !=, <=, >=
- Pre/postfix: ++, --
- Unary: !, -, +

HLSL Operator MSDN Reference

More HLSL Operators

- Bitwise: ~, <<, >>, &, |, ^
  <<=, >>=, &=, |=, ^=
- Array: [index]
- Cast: (type)
- Structure member: .

Per-Component Operators

- Many operators are per-component
  - Operator occurs independently for each component in a multi-component variable (vector or matrix)

- Examples:
  - `int4 data = int4(1, 2, 3, 4);`
  - `int4 result = data * 5; // -> (5, 10, 15, 20)`

  - `float4 red = float4(1, 0, 0, 1);`
  - `float4 darkRed = red * 0.5f;`

Optimizing Per-Component Math

- The following line can be optimized:

```
float4 answer = float4(1,1,1,1) * 5 * 10;
```

  - Results in 8 multiplications

- Optimized version:

```
float4 answer = float4(1,1,1,1) * (5 * 10);
```

  - Results in 5 multiplications

Registers & Semantics

Registers

- Where data is stored during the execution of a given shader
  - Similar to registers in a CPU
- Individual items (constant buffers, samplers, etc) are *always* bound to registers
- Can optionally be bound to specific registers
  - Using the *register* keyword
  - Basically an index

Why Use Registers?

- A shader could use multiple textures
- CPU must bind those textures to the shader
  - For example: PSSetShaderResources()
  - Requires an index (called StartSlot)
- How does the CPU connect a texture to a specific texture register in a shader?
  - *Register* keyword allows you to specify the index

Which Items Have Registers?

- Major register prefix & corresponding type:
  - b – constant buffer registers
  - t – texture registers
  - s – sampler registers
- Items must always use corresponding type
  - Or shader won't compile
  - Example: Textures must always use "t" prefix
- Register keyword allows you to specify the index of each item defined in the shader

Register Example

```
// Registers require a prefix and an index

// Samplers – Must use “s” prefix
sampler LinearWrap : register(s0);
sampler PointClamp : register(s1);

// Textures – Must use “t” prefix
Texture2D BaseColor : register(t0);
Texture2D NormalMap : register(t1);
Texture2D SpecMap   : register(t2);
```

Shader Parameter Semantics

- Semantics are strings attached to shader input or output parameter variables
- Specifies the intended usage of a parameter
- All variables passed between shader stages are required to have semantics
  - Including variables in input or output structs
- Semantics MSDN Reference

Vertex/Pixel Shader Semantics

- Not all semantics are recognized everywhere
- Different shaders support different semantics
  - Different semantics for input & output too
  - See link on previous slide for specifics
- Semantic names can have an optional index
  - `TEXCOORD` – A texture coordinate (uv coord)
  - `TEXCOORD0` – Same as TEXCOORD
  - `TEXCOORD1` – Another texture coordinate

Semantic Examples

- Many semantics have existed since DirectX 9
  - POSITION
  - COLOR
  - TEXCOORD
  - NORMAL

- Some semantics are DX 10 and newer only
  - Example: SV_Position
  - Called "System Value" semantics
  - Begin with SV_ (so there are no naming conflicts)

Semantic Example

```
struct VSInput
{
    float3 pos : POSITION;
    float2 uv  : TEXCOORD;
};

struct VSOutput
{
    float4 pos   : SV_POSITION; // DX10 and above
    float2 uv    : TEXCOORD0;
    float  other : TEXCOORD1;
};
```

Flow Control

Flow Control

- Determines which block of statements to execute next
- HLSL supports standard conditional and loop control statements
  - if / else if / else
  - switch / case / default
  - for / while / do-while
  - break / continue
- Flow control MSDN Reference

Branching

- Sequential statements are executed in order
- Unless a *branch* statement is encountered
  - If statements
  - Loop conditions
- Branching causes different statements to potentially execute based on a condition

Branching Performance

- Potential performance issues!

- Normal code execution:
  - Future instructions are preemptively fetched
  - Basically, upcoming code is being “prepared” early

- When a branch is encountered:
  - Which statement comes next?
  - Not known until branch is evaluated!
  - Processor must wait until current code is done before “preparing” next set of instructions

CPU vs. GPU Branching

- CPU's have branch predictors
  - "Guesses" which branch might be taken
  - Begins fetch and speculative execution
  - If guess was correct – speed up!

- GPU's don't have branch predictors
  - Branching can potentially cause performance issues
  - Unless branch cost is cheaper than code execution

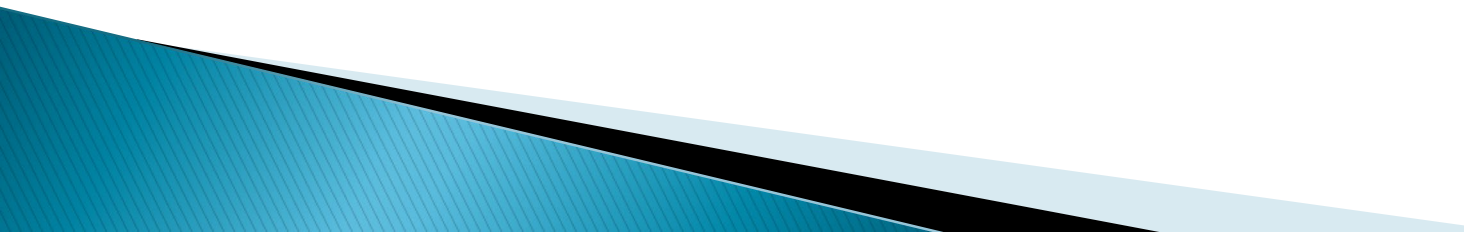
When To Branch?

- Branch when:
  - Cost of branching is less than cost of code block
  - Branch potentially skips most/all of shader
  - Shader performance isn't a high priority
  - Learning HLSL
- Don't branch when:
  - Code can be rewritten without a branch
  - Branching can be replaced by built-in functions
- Shader Optimization MSDN Reference

Branching & Loops

- Loops contain conditions
  - Determine when the loop ends
  - This is a branch!
  - Potential performance issue again!

- HLSL compiler can *unroll* some loops
  - Removes part or all of the loop itself
  - Replaces it with copies of the code block



Unrolling Loop Example

- Loop version:

```
for( int i = 0; i < 3; i++ )
    DoSomething(i);
```

- Unrolled version:

```
DoSomething(0);
DoSomething(1);
DoSomething(2);
```

HLSL Loop Attributes

- Compiler can unroll some loops automatically
- Unrolling isn't always desirable
  - Unrolling results in more instructions
  - Can potentially make shaders "too long" to compile
- Optional loop attributes control compilation:
  - `[unroll]` – Unrolls loop (if possible)
  - `[loop]` – Ensures loop actually loops

Loop Attribute Examples

```
[unroll]
for( int i = 0; i < 10; i++ )
    DoSomething(i);

[loop]
for( int i = 0; i < 10; i++ )
    DoSomething(i);
```

- For Loop MSDN Reference

Intrinsic Functions

Intrinsic Functions

- HLSL has many built-in (intrinsic) functions
  - Mostly math-related
  - Some only work in certain shader types

- Some functions are per-component
  - They can be used with scalars, vectors or matrices
  - Will perform their function on the whole input
  - Output type will match input type

- List of HLSL Intrinsic Functions

Major Math Functions

- Basic math:
  - `abs, floor, ceil, frac, sign`
  - `round, trunc, sqrt, pow`
- Trig functions:
  - `sin, cos, tan, sincos`
  - `asin, acos, atan, atan2`

Vector & Matrix Functions

- Vector functions:
  - `dot, cross`
  - `length, distance`
  - `normalize`
- Matrix functions:
  - `mul`

Clamping Values

- `clamp( data, min, max )`
  - Clamps *data* between *min* and *max*
  - If data < min, it becomes min
  - If data > max, it becomes max
  - Otherwise it doesn't change

- `saturate( data )`
  - Clamps *data* between 0.0 and 1.0
  - Same as `clamp( data, 0.0, 1.0 )`

Interpolation

- `lerp( v1, v2, percent )`
  - Interpolates between v1 and v2 based on percent
  - Basically this function: v1 + percent * (v2 – v1)
  - A percent of 0.0 gives v1
  - A percent of 1.0 gives v2

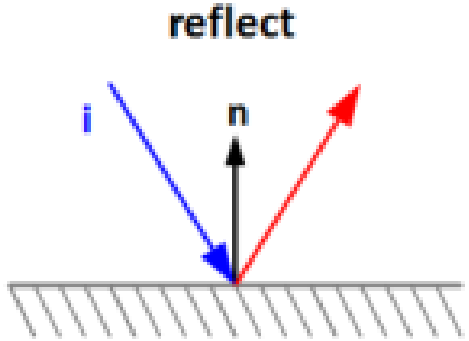
- `smoothstep( min, max, data )`
  - A smooth Hermite interpolation between 0 & 1
  - If data < min, returns 0
  - If data > max, returns 1
  - Otherwise returns a value between 0 and 1

Choices

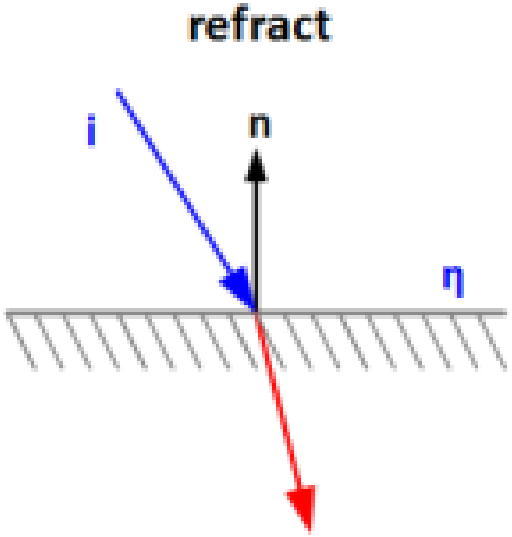
- `step( v1, v2 )`
  - Returns either 0 or 1
  - Returns 1 if v1 >= v2
  - Returns 0 otherwise
- `max( v1, v2 )`
  - Returns either v1 or v2, whichever is greater
- `min( v1, v2 )`
  - Returns either v1 or v2, whichever is smaller

Vector Helpers

- `reflect( incident, normal )`
  - **Incident** is the incoming vector
  - **Normal** is the surface normal



- `refract( indicent, normal, refractionIndex )`
  - **Indicent** is the incoming vector
  - **Normal** is surface normal
  - **RefractionIndex** defines how much the vector “bends”



Per-Component Comparisons

- HLSL has 2 functions that look for non-zero values in a multi-component variable

- all(data)
  - Returns true if *all* components of *data* are non-zero

- any(data)
  - Returns true if *any* components of *data* are non-zero

Per-Component Comparisons

- Comparison operators can return multiple booleans
  - If both sides are vectors, for example

```
float4 big   = float4(5,5,5,0);
float4 small = float4(1,1,1,9);

bool4 result = big > small;
// Result is (true, true, true, false)
```

Per-Component Comparisons

- But if statements only work with single booleans

```
float4 big   = float4(5,5,5,0);
float4 small = float4(1,1,1,9);
if( big > small ) // ERROR: Won't compile!

// This works (but there is an easier way)
if( big.x > small.x && big.y > small.y &&
    big.z > small.z && big.w > small.w )

// Cleaner – All components must be true
if( all( big > small ) )
```

Custom Functions

Custom Functions

- Individual shaders are simply functions
  - Vertex, pixel, etc.
- Can write custom helper functions too
- Often helper methods are put in external files
  - Can be #include'd into other .hlsl files
- Function MSDN Reference

Function Parameters

- Parameters are “in” params by default
  - Represent data being passed in to the function
  - Standard parameter behavior
- Parameters can also be “out” parameters
  - Represent data being passed back out
  - Allows a function to “return” more than one value
- Also supports “inout” parameters
  - Data coming in, which can change and is passed back out at the end of the function

Function Performance

- All custom functions in HLSL are “inline”
  - Basically the function body is pasted into the shader
  - Rather than actually “calling” the function
- No performance hit when using functions!
  - No function call overhead since they’re inline
- Still adds to the instruction count
  - Could possibly generate a shader that is “too long”

Example

A simple example of the pieces necessary for a basic directional light

Example – Vertex Shader

Example – VS Constant Buffer

```
// The name of the cbuffer is largely
// irrelevant. What matters is the
// register and the data inside!
cbuffer DataFromCPU : register(b0)
{
    matrix World;
    matrix WorldInverseTranspose;
    matrix View;
    matrix Projection;
}
```

Example – VS Input Struct

```
// This struct represents the data of a
// single vertex in our vertex buffer.
// This must also match our Input Layout
struct VS_Input
{
    float3 localPosition : POSITION;
    float3 normal        : NORMAL;
    float2 uv            : TEXCOORD;
};
```

Example – VS Output Struct

```
// Represents the data coming out of the
// Vertex Shader, which must also match
// the input of the Pixel Shader
struct VS_Output
{
    float4 screenPosition : SV_POSITION;
    float3 normal         : NORMAL;
    float2 uv             : TEXCOORD;
};
```

Example – Vertex Shader Signature

```
// Beginning of the vertex shader itself.
// Its return type is a struct.
// Its parameter is also a struct.
VS_Output main( VS_Input input )
{
    // Create eventual output variable
    VS_Output output;

    // Continues on next slide...
```

Example – VS – Screen Position

```
// Calculate World View Projection matrix
// (Backwards because of DXMath / HLSL transpose issue)
matrix wvp = mul(Projection, mul(View, World));

// Use final matrix to calculate the
// vertex's screen position
output.screenPosition =
    mul(wvp, float4(input.localPosition, 1.0f));

// Continues on next slide...
```

Example – VS – Normal & UV

```
    // Vertex normal must be transformed too
    output.normal =
        mul((float3x3)WorldInverseTranspose, input.normal);

    // Pass the uv through, since it comes
    // from the vertex but isn't used here
    output.uv = input.uv;

    // End of vertex shader – return data
    return output;
}
```

Example – Pixel Shader

Example – PS Input Struct

```
// Represents the input data of the Pixel Shader, which
// must match the output struct of the Vertex Shader
struct PS_Input
{
    float4 screenPosition : SV_POSITION;
    float3 normal         : NORMAL;
    float2 uv             : TEXCOORD;
};
```

Example – PS Helper Struct

```
// Represents a single directional light
// with ambient and diffuse color
struct DirectionalLight
{
    float4 AmbientColor;
    float4 DiffuseColor;
    float3 Direction;
};

// Will be part of the constant buffer
```

Example – PS Constant Buffer

```
// We need to get the light data from
// the CPU side, which means passing
// it in through a constant buffer
cbuffer PSDataFromCPU : register( b0 )
{
    DirectionalLight Light;
};
```

Example – Other PS Variables

```
// The texture for the model
Texture2D DiffuseTexture : register(t0);

// Sampler which describes how to pull
// pixels from a texture
SamplerState TrilinearSampler : register(s0);
```

Example – Pixel Shader Signature

```
// Beginning of Pixel Shader itself
// Return type is float4 (a color)
// Parameter is PS_Input struct
float4 main(PS_Input input) : SV_TARGET
{

    // Continues on next slide...
```

Example – PS – Basic Lighting

```
// Normalize the input normal
float3 normal = normalize(input.normal);

// Invert the light's direction
float3 lightDir = -Light.Direction;

// Calculate the diffuse light factor
float lightAmount = dot(normal, lightDir);

// Clamp light factor between 0 – 1
lightAmount = saturate(lightAmount);
```

Example – PS – Texture

```
// Sample the texture using the sampler
// and the interpolated uv coords
float4 textureColor =
  DiffuseTexture.Sample(TrilinearSampler, input.uv);
```

Example – PS – Final Color

```
    // Put it all together!  First figure
    // out final diffuse light color
    float4 lightColor = lightAmount * Light.DiffuseColor;

    // Add in ambient light
    lightColor += Light.AmbientColor;

    // Return texture modulated by light
    return textureColor * lightColor;
}
```