VU Entwurf und Programmierung einer Rendering-Engine

Domain Specific Languages and Shader Systems for Rendering Engines

- What are DSLs in rendering systems?
- Shader system designs
- Shader DSLs
- Demo/Discussion

Domain Specific Languages (DSLs)?

Definition

- A domain specific language (DSL) is a computer language designed for application in a specific domain
- DSLs are used to effectively describe something, and to allow for abstraction and optimization (compiler techniques, semantic understanding)
- Examples: HTML, SQL, MATLAB (matrix arithmetic), Torch (tensor operations), Renderman, various game scripting languages, …

https://negativespace.co/wp-content/uploads/2017/10/negative-space-code-html-dark-background-Mian-Shahzad-Raza-thumb-1.jpg

Software Design Patterns *vs* (Domain Specific) Language-based approaches

- Software engineering has its own language
    - Words like visitor, state pattern, aggregation, inheritance, dependency injection, singleton
    - For some rendering engine modules, this language is adequate
- Rendering touches on many specific domains
    - **Animation/Storytelling**: when designing animations, we don’t care about classes.
    - **Shaders**
    - **Scene description**, e.g. RenderMan uses own language to model scenes
- Two reasons to implement a DSL
    - A language may be a **better description** than software concepts
    - We would like to do **optimizations** based on the domain-specific semantics

```
PixelVariance .007
Format 500 500 1.0
ShadingRate 1.0
Projection "perspective" "fov" [ 39.14625166082039 ] # lens 45.0, aspect 1.0
Rotate 180 0 1 0 # right handed
Scale -1 1 1 # right handed
```

Motivation 1: Complexity Management

Shading Pipelines have many functionalities (light types, visibility, filtering), using differently encoded in-/outputs (forward/deferred pipeline), and have interchangeable alternatives (microfacet/fresnel BRDF, shadow masking)

Geometry	Light	Visibility	Filtering	BRDF	Post Effects
Displacement	Point	Cubemap	PCF	Diffuse	Fog
Tessellation	Spot	Perspective	Poisson	Phong	Scattering
Animation	Area	Paraboloid	PCSS	Ward	Tonemap
Instancing	Directional	Cascaded	CSS	GGX	
	Photometric	Imperfect	…	…	
	Disk	Stochastic			
	...	...			

	Visibility	Filtering	Area Light Approx		
	+Image-based Lighting			BRDF	
	+Global Illumination			BRDF	

Motivation 2: Performance and Feature Abstraction

- Graphics APIs keep evolving to have increasingly specialized capabilities for specific rendering scenarios.
  - Inputs grouped by update frequency (descriptor sets/tables)
  - Hardware Instancing
  - Stereo rendering
  - Input layout unification, to make shader switches less costly
  - Code re-use on CPU/GPU

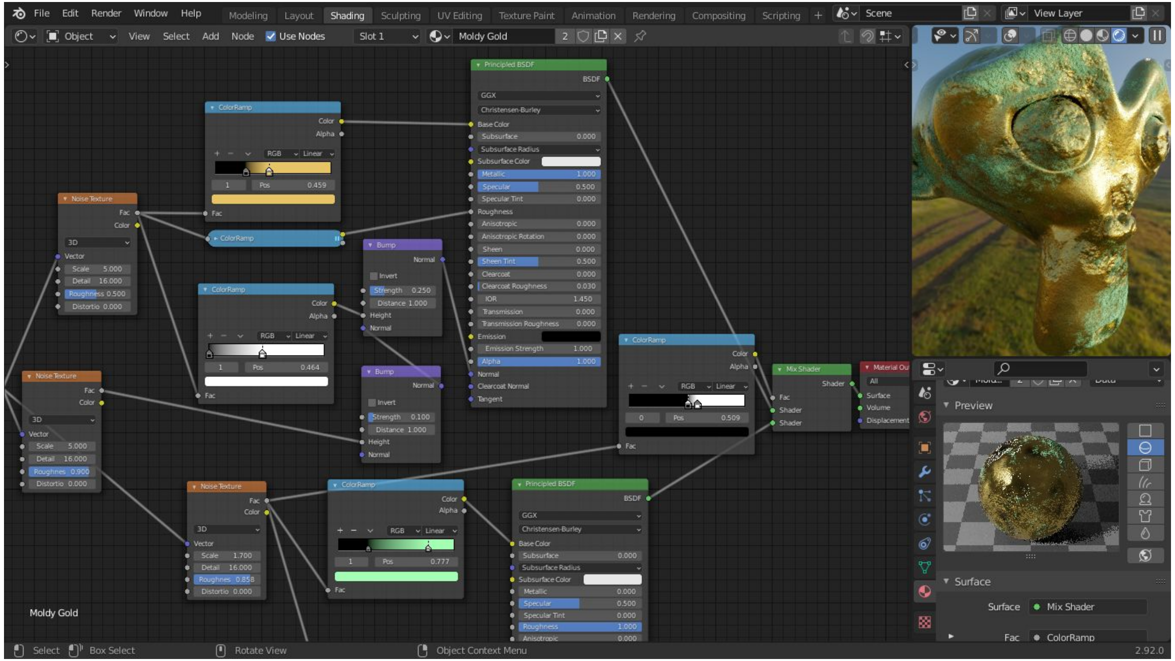
```
for( sporadically )
{
        Bind Descriptor Set #0
        for( the entire scene )
        {
                Bind Descriptor Set #1
                for( each object in the scene )
                {
                        Bind Descriptor Set #2
                        Do the drawing
                }
        }
}
```

Vulkan spec, updating inputs at different frequencies

https://www.trustedreviews.com/explainer/what-is-virtual-reality-2940543

Motivation 3: “Programmability”, Frontend

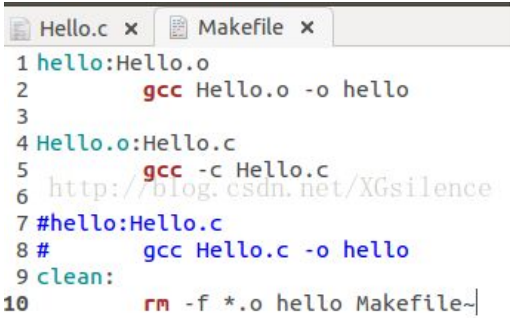
- Artists/API consumers should be able to make bespoke modifications and arrangements (sometimes at runtime)
- There are many deploy targets (opengl, webgl, spir-v, hlsl, …) and hardware capabilities we may need to respect

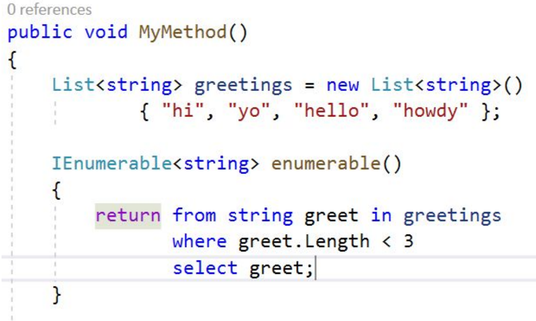


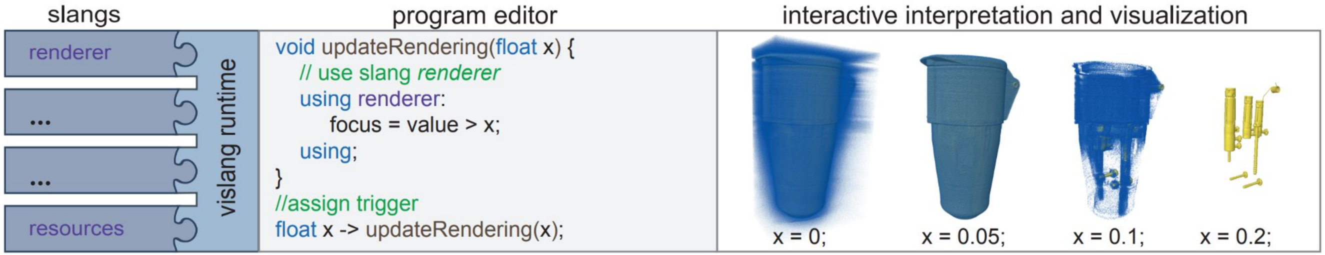
Blender wiki, shader editor

Types of DSLs

- External DSL:
  - Use lexer, parser, program analysis and code generation for specific problem
  - However, there are many tools helping here (e.g. LLVM, YACC, spoofax)
  - Advantage: freedom in language design
- Embedded DSL:
  - Language expressed in host language
    - e.g. by using implicit conversions, higher order functions etc
  - Advantages
    - fluently embedded in host program
    - can interact with embedding environment







[Rautek et al. 2014]
https://www.cg.tuwien.ac.at/research/publications/2014/Rautek_Peter_2014_VSA/

Embedded Domain Specific Languages

- Two implementation approaches
  - Interpretation
    - Non performance critical tasks: e.g. DSL for animation or story-telling
  - Code Generation
    - The DSL expression tree is traversed in order to generate native code
    - This is a very common technique
    - Compiling embedded languages, [Elliott 2003]

```
var transformed = input.DoByVertex(v => {
  v.Position = Uniform.ModelViewProjTrafo * v.Position;
  v.Normal = Uniform.NormalTrafo * v.Normal;
  v.WorldPosition = Uniform.ModelTrafo * v.Position;
  return v;
});
```

Cosmo [9] compiles shader code from embedded language

Implementation of Embedded DSLs

- Deep embedding
  - Host language builds an expression tree
  - Expression tree can be analyzed and optimized
  - In order to be used for
    - Code generation
    - Interpretation
- Shallow embedding
  - No explicit expression tree built
  - Optimizations are hard

```
with sh.function('add', sh.Float4)(a = sh.Float4, b = sh.Float4):
    sh.return_(sh.a + sh.b)
```

```
float4 add(float4 a, float4 b)
{
        return (a + b);
}
```

Source [6], metashade builds expression tree and compiles it to shader code

```
public abstract class Exp
{
    public abstract string Compile();
    public static Exp operator +(Exp a, Exp b)
    {
        return new Add(a, b);
    }
    public static implicit operator Exp(int d)
    {
        return new Lit(d);
    }
}
public class Lit : Exp
{
    int value;
    public Lit(int l) { value = l; }
    public override string Compile() { return string.Format("{0}", value); }
}
public class Add : Exp
{
    Exp left; Exp right;
    public Add(Exp l, Exp r) { left = l;  right = r; }
    public override  string Compile()
    {
        return string.Format("({0}+{1})", left.Compile(), right.Compile());
    }
}
```

```
var add1 = new Add(new Lit(1), new Lit(2));
var add2 = new Add(add1, add1);
Console.WriteLine(add2.Compile());
```

=> ((1+2)+(1+2))

Use caching for common subexpressions

Nice syntax using Implicit conversion

Operator overload

```
Exp add3 = 2 + 2;
Exp add4 = add3 + add3;
Console.WriteLine(add4.Compile());
```

Embedded DSL provides abstraction for free

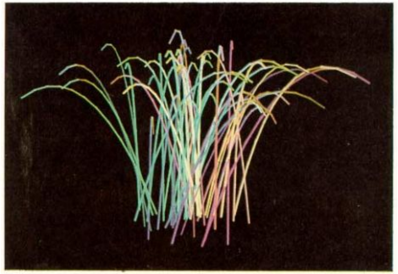
- Embedded DSL has two execution times
  - 1. When the expression is constructed
  - 2. When the expression is interpreted or its compiled variant is executed
- This property gives us abstraction for free
- Parameters, Virtual Functions, Lambdas etc, are evaluated when constructing the expr. Tree
- Rompf et al. 2010, Lightweight Modular Staging
- Seitz et al. 2019, Staged Metaprogramming for Shader System Development

```
local MaterialSystem = require("MaterialSystem")
...
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  ...
  uniform LightData {
    @UIType(Slider3) lightDirection : vec3
  }
  ...
  fragment code
    ...
    color = [MaterialType:eval()](shadingData)
    return color * max(0, dot(shadingData.normal, lightDirection
  end
}
```

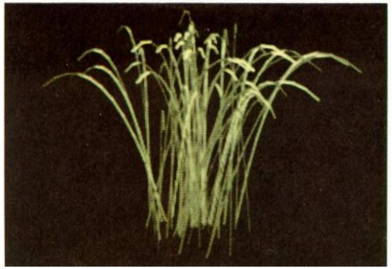
DSLs for rendering systems

Shade Trees

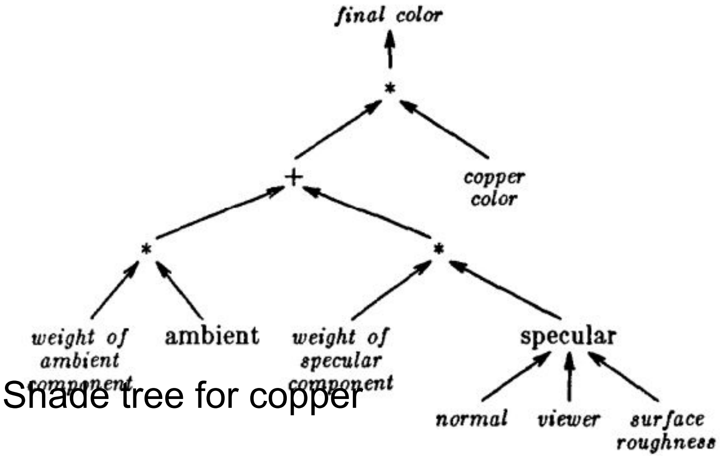
Grass Normal Texture



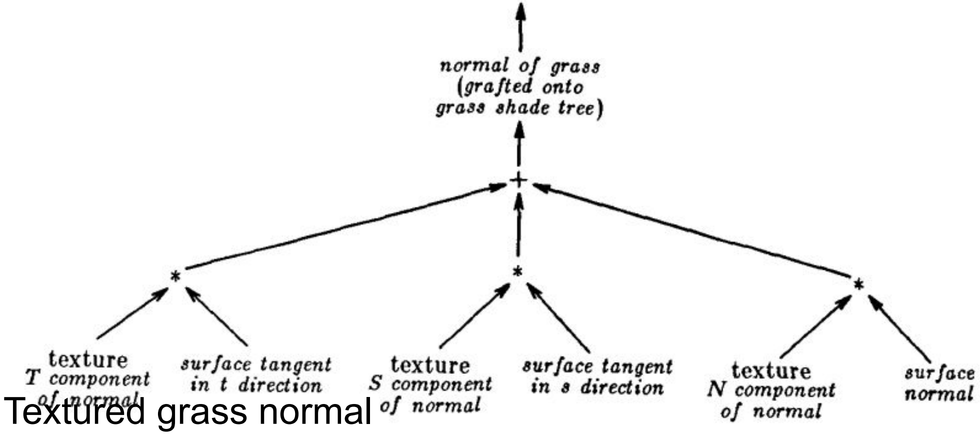
Grass with textured normals



- Cook’s shade trees [Cook 1984] Disney
- Idea: no single shading model appropriate for all use cases
- Independent aspects like lighting, surface & atmospheric into separate modules
- Purely functional forms expression tree (no loops, assignments)

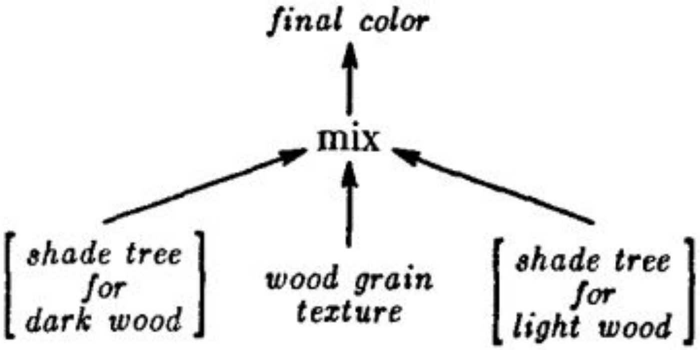


Shade tree for copper



Textured grass normal

Shade Trees: Language vs API



Language approach:

```
finalColor = mix(wood(),
                 woodGrain.Sample2d(tc),
                 lightWood)
```

Software design/API approach:

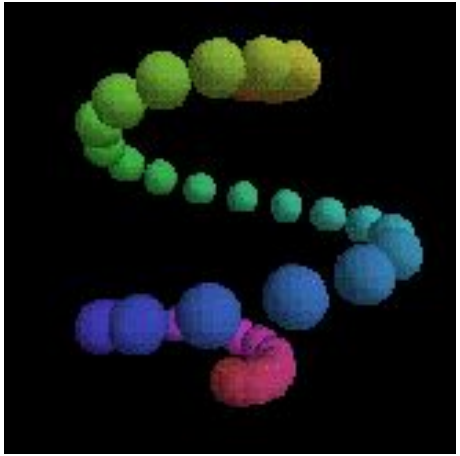
```
ExpressionBuilder e = new ExpressionBuilder();
e.CreateNewMix(e.Sample(this.woodGrain,tc));
e.Input1 = base.Wood
e.Input2 = base.lightWood
```

Impractical

Functional Reactive Animation

FRAN, [Elliott 1997]

```
spiralTurn = turn3 zVector3 (pi*time) (unionGs (map ball [1 .. n]))
  where
    n = 40
    ball i  = withColorG color (                    // colorize it
                move3 motion (                      // move it by motion
                  stretch3 0.1 sphereLowRes  // scale it
               ))
     where
       motion = vector3Spherical 1.5 (10*phi) phi
       phi       = pi * fromInt i / fromInt n
       color    = colorHSL (2*phi) 0.5 0.5
```

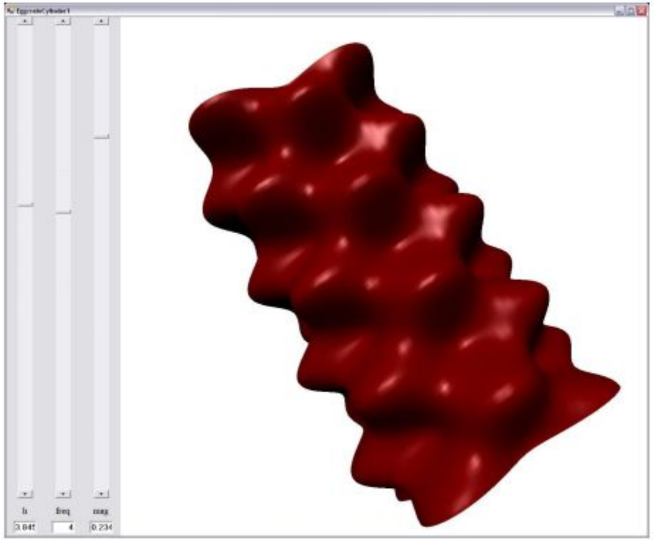


Vertigo

- [Elliot 2004], Embedded DSL for scene description in Haskell, Code generation generates Vertex/Fragment shader (assembly)

type Surf = R^2 → R^3
sphere :: Surf
sphere (u, v) = (cos θ ·sin φ, sin θ ·sin φ, cos φ)
    where θ = 2 · π · u φ = π · v
displace :: Surf → HeightField → Surf
displace surf field = surf + field · normal surf

normal :: Surf → Surf
normal = normalize ◦ cross ◦ derivative



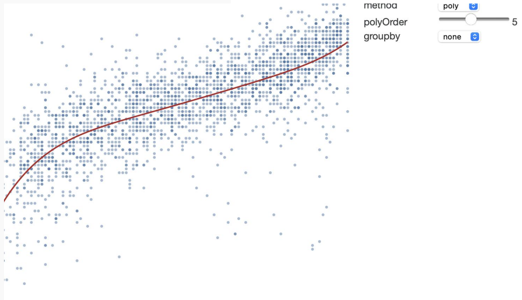
eggcrateCylinder 3.8 4.0 0.23

Visualization DSLs

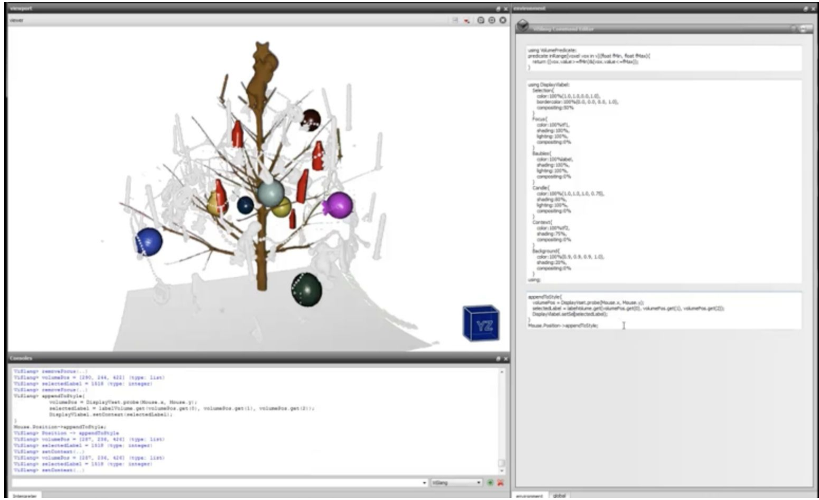
- Include interactivity and data wrangling

VEGA: https://vega.github.io/vega/examples/regression/

```
{
  "$schema": "https://vega.github.io/schema/vega/v5.json",
  "description": "A scatter plot with trend line calculated via user-configurable regression methods.",
  "padding": 5,
  "width": 500,
  "height": 500,
  "autosize": "pad",

  "signals": [
    {
      "name": "method", "value": "linear",
      "bind": {"input": "select", "options": [
        "linear", "log", "exp", "pow", "quad", "poly"
      ]}
    },
    {
      "name": "polyOrder", "value": 3,
      "bind": {"input": "range", "min": 1, "max": 10, "step": 1}
    },
    {
      "name": "groupby", "value": "none",
      "bind": {"input": "select", "options": ["none", "genre"]}
    }
  ],

  "data": [
    {
      "name": "movies",
      "url": "data/movies.json",
      "transform": [
        {
          "type": "filter",
          "expr": "datum['Rotten Tomatoes Rating'] != null && datum['IMDB Rating'] != null"
        }
      ]
    },
    {
      "name": "trend",
      "source": "movies",
      "transform": [
        {
          "type": "regression",
          "groupby": [{"signal": "groupby === 'genre' ? 'Major Genre' : 'foo'"}],
          "method": {"signal": "method"},
          "order": {"signal": "polyOrder"},
          "extent": {"signal": "domain('x')"},
          "x": "Rotten Tomatoes Rating",
          "y": "IMDB Rating",
          "as": ["u", "v"]
```



ViSlang
https://www.cg.tuwien.ac.at/research/publications/2014/Rautek_Peter_2014_VSA/



Lambdacube 3D

- Entire rendering pipeline is one DSL written in Haskell
- Rendering pipeline is abstracted as composable functions
- CPU and GPU code is the same

http://lambdacube3d.com/

```haskell
makeFrame (time :: Float)
          (vertexstream :: PrimitiveStream Triangle (Vec 4 Float,Vec 2 Float))

          = imageFrame (emptyDepthImage 1, emptyColorImage navy)
  `overlay` fragments
  where
   projmat = perspective 0.1 100.0 (30 * pi / 180) 1.0
          .*. lookat (V3 3.0 1.3 0.3) (V3 0.0 0.0 0.0) (V3 0.0 1.0 0.0)
          .*. rotMatrixY (pi / 24.0 * time)

   sampler = Sampler LinearFilter MirroredRepeat $ Texture2DSlot "Diffuse"
   fragments =
          vertexstream
        & mapPrimitives (\(x,uv) -> (scale 0.5 (projmat *. x), uv))
        & rasterizePrimitives (TriangleCtx CullNone PolygonFill NoOffset LastVertex) ((Smooth))
        & mapFragments (\((x)) -> ((texture2D sampler x)))
        & accumulateWith (DepthOp Less True, ColorOp NoBlending (V4 True True True True))

main = renderFrame $
   makeFrame (Uniform "Time")
             (fetch "stream4" (Attribute "position4", Attribute "vertexUV"))
```

Shader Systems

GLSL Example

Render everything white

```
uniform UBO{
    mat4 MVPMat;
};

in vec4 Position;
void Vertex()
{
    vec4 pos = MVPMat* Position;
    gl_Position = pos;
}
```

```
out vec4 Color;
void Fragment()
{
    Color = vec4(1,1,1,1);
}
```

GLSL Example: Texturing

Render everything textured - semantic change leads to many code changes

```
uniform UBO{
    mat4 MVPMat;
};

in vec4 Position;
in vec2 TexCoord;
out vec2 fs_TexCoord
void Vertex()
{
    vec4 pos = MVPMat* Position;
    fs_TexCoord = TexCoord;
    gl_Position = pos;
}
```

```
uniform sampler2D tex;

in vec2 fs_TexCoord;
out vec4 Color;
void Fragment()
{
    Color = texture(tex, fs_TexCoord);
}
```

The abstraction pyramid of Shader Systems

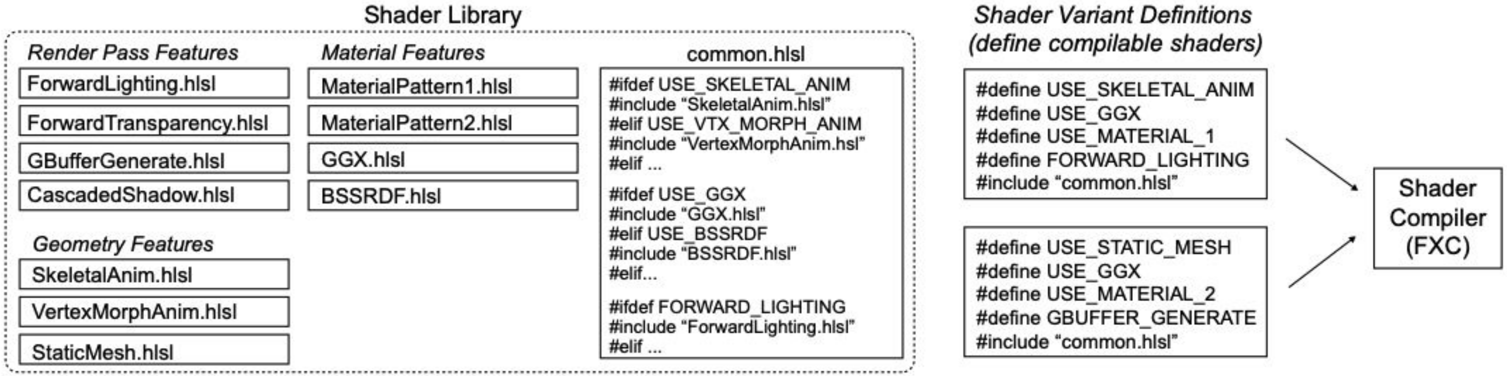
In decreasing order of abstraction:

- interfaces, mixins, OOP approach (ex. Spark)
- Embedded DSL, FP approach (ex. FShade)
- External DSL (ex. Unity ShaderLab)
- Übershader/Ifdef - Metaprogramming
- —--------------------- (Graphics API barrier)
- GLSL, HLSL, WGSL, high level input languages
- intermediate languages (DXIL, SPIR-V)
- ASM

```
// colored vertex lighting
Shader "Simple colored lighting"
{
    // a single color property
    Properties {
        _Color ("Main Color", Color) = (1,.5,.5,1)
    }
    // define one subshader
    SubShader
    {
        // a single pass in our subshader
        Pass
        {
            // use fixed function per-vertex lighting
            Material
            {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

https://docs.unity3d.com/560/Documentation/Manual/SL-Shader.html

“Ifdef-style”



Source: [1], Unreal Engine 4 (ca 2015) shader modularization via preprocessor directives

- Shader compiler preprocessor directives used to enable/disable parts of code
- Helps avoid branching in shader code
- Hard to adapt shader inputs, hard to manage attribute passing (or always pass all inputs (e.g. using empty textures if none).
- Difficult to maintain

Übershader

- Write everything into one shader, modularity at runtime (using if-else)
- Requires unified input bindings (e.g. `if(hasTexture){...})`.
- Good match with extensive modern material systems used in computer graphics (ex. OpenPBR)
- Limited feature set becomes prohibitive
- No real tooling

Unity-style Macros

- Provided macros are “filled in” at runtime by the engine
- Unity’s HLSL/ShaderLab exposes many shader-related features like this (platform specific feature sets, textures/cubemaps, shadow mapping,)
- Be careful with definitions, little IDE support

https://docs.unity3d.com/6000.0/Documentation/Manual/SL-BuiltinFunctions.html

Macro:	Use:
`UNITY_DECLARE_SHADOWMAP(tex)`	Declares a shadowmap Texture variable with name “tex”.
`UNITY_SAMPLE_SHADOW(tex,uv)`	Samples shadowmap Texture “tex” at given “uv” coordinate (XY components are Texture location, Z component is depth to compare with). Returns single float value with the shadow term in 0..1 range.
`UNITY_SAMPLE_SHADOW_PROJ(tex,uv)`	Similar to above, but does a projective shadowmap read. “uv” is a float4, all other components are divided by .w for doing the lookup.

Sh - Shader Metaprogramming language

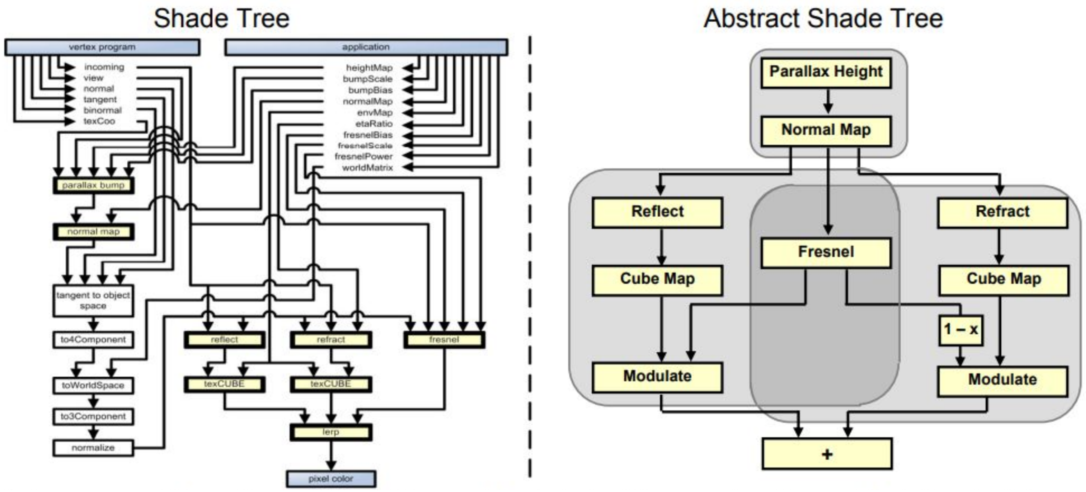
- Embedded DSL
- Support Template programming
- Additional degree of abstraction
- Shader code generation is delayed, allowing for some code analysis

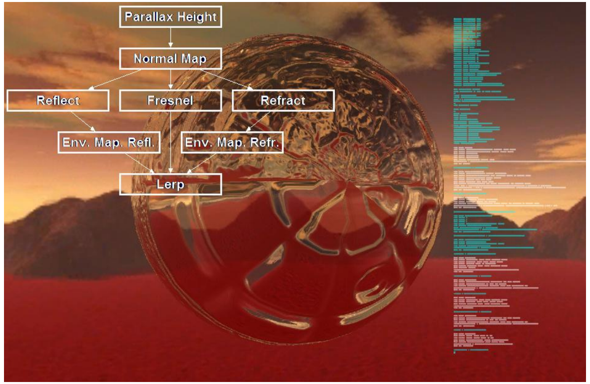
```
class BaseShader {
  public:
    static ShMatrix4x4f VD;            // VCS to DCS
    static ShMatrix4x4f MV;            // MCS to VCS
    static ShMatrix4x4f MD;            // MCS to DCS
    static ShMatrix4x4f inverse_MV;    // MCS from VCS
    ShProgram vertex_shader;
    ShProgram fragment_shader;
    BaseShader();
    void bind();
    virtual void init() = 0;
};
template <int NLIGHTS>
class PointLightShader: public BaseShader {
  public:
    static ShPoint3f light_position[NLIGHTS];
    static ShColor3f light_color[NLIGHTS];
    PointLightShader();
};
template <int NLIGHTS>
class BlinnPhongShader: public PointLightShader<NLIGHTS> {
  public:
    ShColor3f ks;      // specular color
    ShColor3f kd;      // diffuse color
    ShAttrib1f exp;    // exponent
    BlinnPhongShader();
    virtual void init();
};
```

Source: [2], Sh metalanguage (2005) generates shader code

Abstract Shade Trees

- [Morgan McGuire et al. 2006]
- Semantic annotations (TEXCOORD, TANGENT,..)

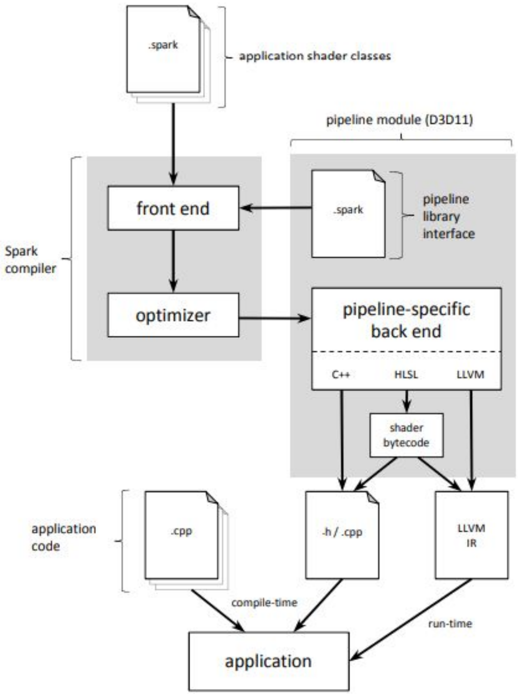




- [Morgan McGuire et al. 2006]

Spark

- **Foley 2011 [5]**
- External DSL
- **Pipeline Shader** concept
- Composability on class level (mixins)
- Frequency annotations
- No shader stages = single function => easy to compose
- Compiler maps to shader stages



- [5]

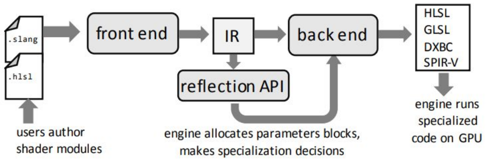
```
shader class Base extends D3D11DrawPass
{
    input @Uniform float4x4     modelViewProjection;
    input @Uniform uint         vertexCount;
    input @Uniform SamplerState linearSampler;

    // Stream of vertices in memory
    struct PNuv { float3 P; float3 N; float2 uv; }
    input @Uniform VertexStream[PNuv] vertexStream;

    // Bind number and type of primitives to draw
    override IA_DrawSpan = TriangleList(vertexCount);

    // Per-coarse-vertex - fetch from buffer
    @CoarseVertex PNuv assembled =
        vertexStream(IA_VertexID);
    @CoarseVertex float3 P_base  = assembled.P;
    @CoarseVertex float2 uv = assembled.uv;

    // Declare model-space position to be virtual
    virtual @FineVertex float3 P_model = P_base;

    // Bind clip-space position for rasterizer
    override RS_Position = mul(float4(P_model, 1.0f),
                               modelViewProjection);
}

mixin shader class Displace extends Base
{
    input @Uniform Texture2D[float3] displacementMap;

    // Per-fine-vertex - displace
    @FineVertex float3 disp =
        SampleLevel(displacementMap, linearSampler,
                    uv, 0.0f);

    override P_model = P_base + disp;
}

mixin shader class Shade extends Base
{
    input @Uniform Texture2D[float4] colorMap;

    // Per-fragment - sample color
    @Fragment float4 color = Sample(colorMap,
                                    linearSampler,
                                    uv);
    // Per-pixel - write to target
    output @Pixel float4 target = color;
}

shader class Example extends Displace, Shade {}
```

Slang

- Foley 2018 [4]
- External DSL
- Uses advanced OOP concepts to achieve high expressivity
- Type **extensions**
    - E.g for BRDF-specific light types
- Associated types
    - E.g define BRDF type while defining associated material type

```
struct QuadLight : ILightEnv {
  float3 vertices[4];
  float3 intensity;
  float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,
                             float3 wo) {
    return bxdf.acceptQuadLight(
      this,
      geometry,
      wo);
  }
}
interface IAcceptQuadLight {
  float3 acceptQuadLight(
    QuadLight       light,
    SurfaceGeometry geom,
    float3          wo);
}

extension IBxDF : IAcceptQuadLight {}

extension Lambertian : IAcceptQuadLight {
  float3 acceptQuadLight(QuadLight     light,
                         SurfaceGeometry geom,
                         float3          wo) {
    return albedo * LTC_Evaluate(light, geometry, wo,
      float3x3(1,0,0,  0,1,0,  0,0,1));
  }
}
extension DisneyBRDF : IAcceptQuadLight { ... }
```

Source:[4], retrospective type extensions used for brdf-specific lights



Source:[4], Slang uses reflection to analyze code and reason about parameter layouts

Customizable code generator in MaterialX

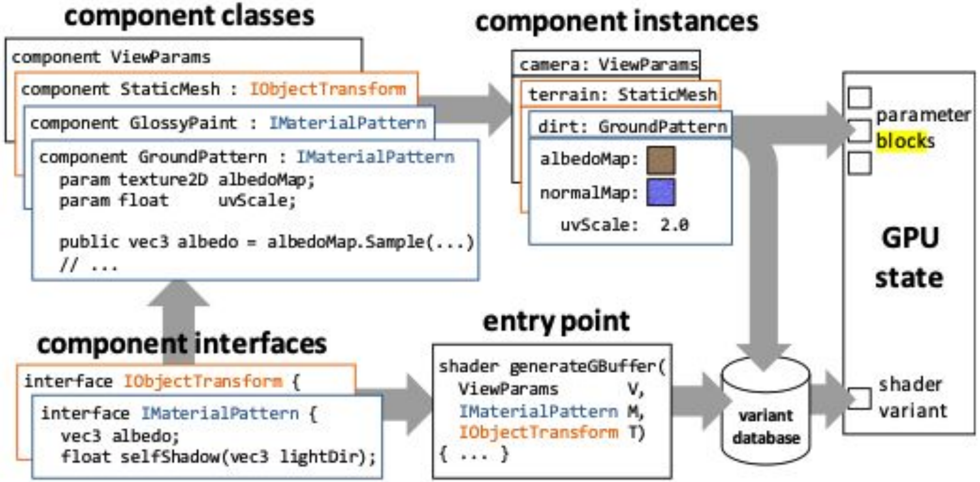
- MaterialX is an exchange format and material spec
- allows for custom shader code by providing access to the code generator
- Provides closure and generator state
- Code decides which lines are emitted
- Very high degree of flexibility

```
void TexCoordNodeGlsl::emitFunctionCall(const ShaderNode& node,
                                        GenContext& context,
                                        ShaderStage& stage) const
{
    const ShaderGenerator& shadergen = context.getShaderGenerator();

    const ShaderInput* indexInput = node.getInput(INDEX);
    const string index = indexInput ? indexInput->getValue()->getValueString() : "0";
    const string variable = "texcoord_" + index;

    DEFINE_SHADER_STAGE(stage, Stage::VERTEX)
    {
        VariableBlock& vertexData = stage.getOutputBlock(HW::VERTEX_DATA);
        const string prefix = vertexData.getInstance() + ".";
        ShaderPort* texcoord = vertexData[variable];
        if (!texcoord->isEmitted())
        {
            shadergen.emitLine(prefix + texcoord->getVariable() + " = i_" + variable, stage);
            texcoord->setEmitted();
        }
    }
```

https://github.com/AcademySoftwareFoundation/MaterialX/blob/main/documents/DeveloperGuide/ShaderGeneration.md

Even Higher Level Languages

- Explicitly design host code encapsulating shading system features
- Can choose which properties to expose to the artist, the engine, the UI, the shader compiler …
- High analysis and abstraction potential
  - ex: engine swaps between single and multipass rendering automatically
  - ex: programmer adds new backend target without having to change existing shaders
- Shader Components[1], Slang[4], Spark[5], Open Shading Language[8], FShade



Source: [1], Shader Components (2017) maintains shader variant database constructed using interfaces and classes

FShade

Transform: FShade

```
type Vertex =
    {
        [<Position>]    pos        : V4d
        [<Normal>]      n          : V3d
        // ...
    }

let transform (v : Vertex) =
    vertex {
        return {
            pos = uniform.MVPMat * v.pos
            n = uniform.NormalMat * v.n
            // ...
        }
    }
```

White: FShade

```
type Vertex =
    {
        [<Color>]       color   : V4d
    }


let white (v : Vertex) =
    fragment {
        return { color = V4d( 1,1,1,1) }
    }
```

Transform & White: GLSL output

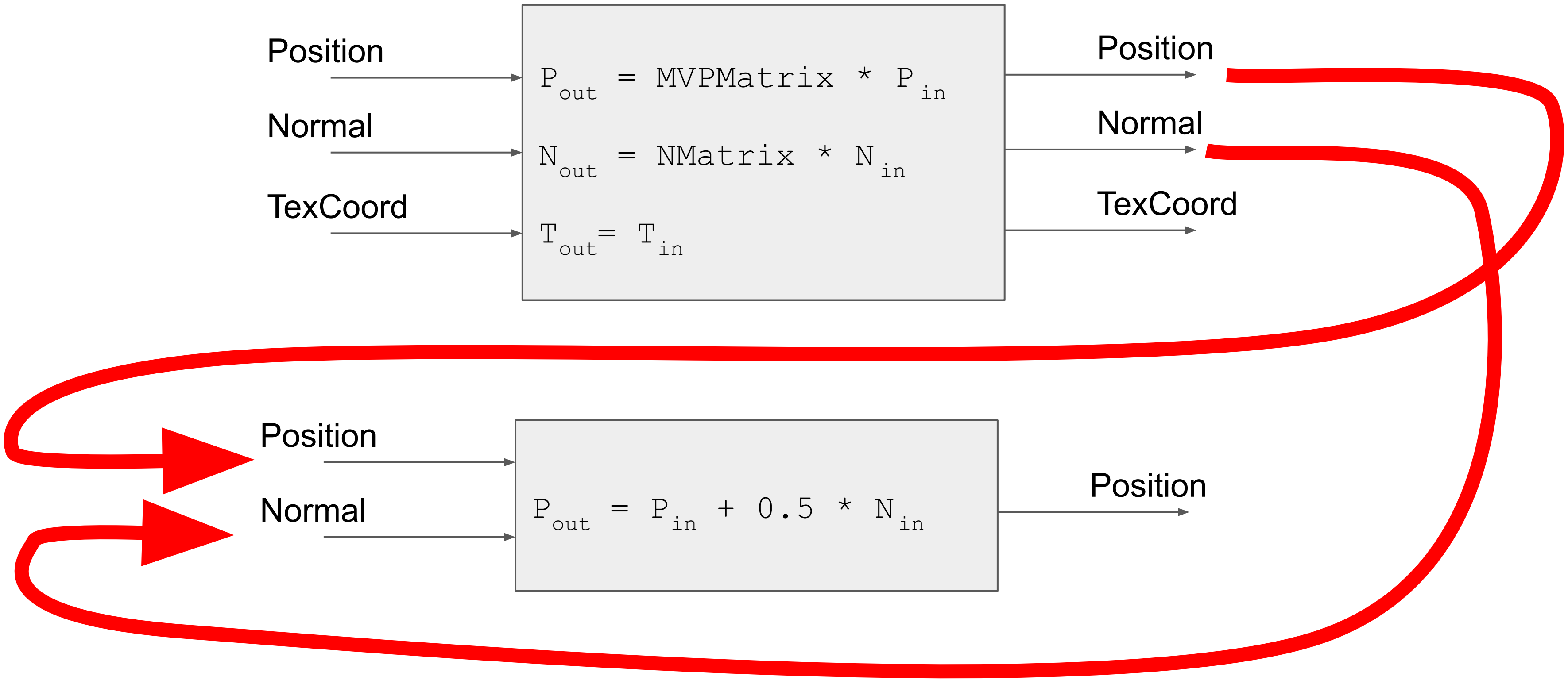
```
compose [ transform; white ]
```

```
uniform UBO{
    mat4 MVPMat;
};

in vec4 Position;
void Vertex()
{
    vec4 pos = MVPMat * Position;
    gl_Position = pos;
}
```

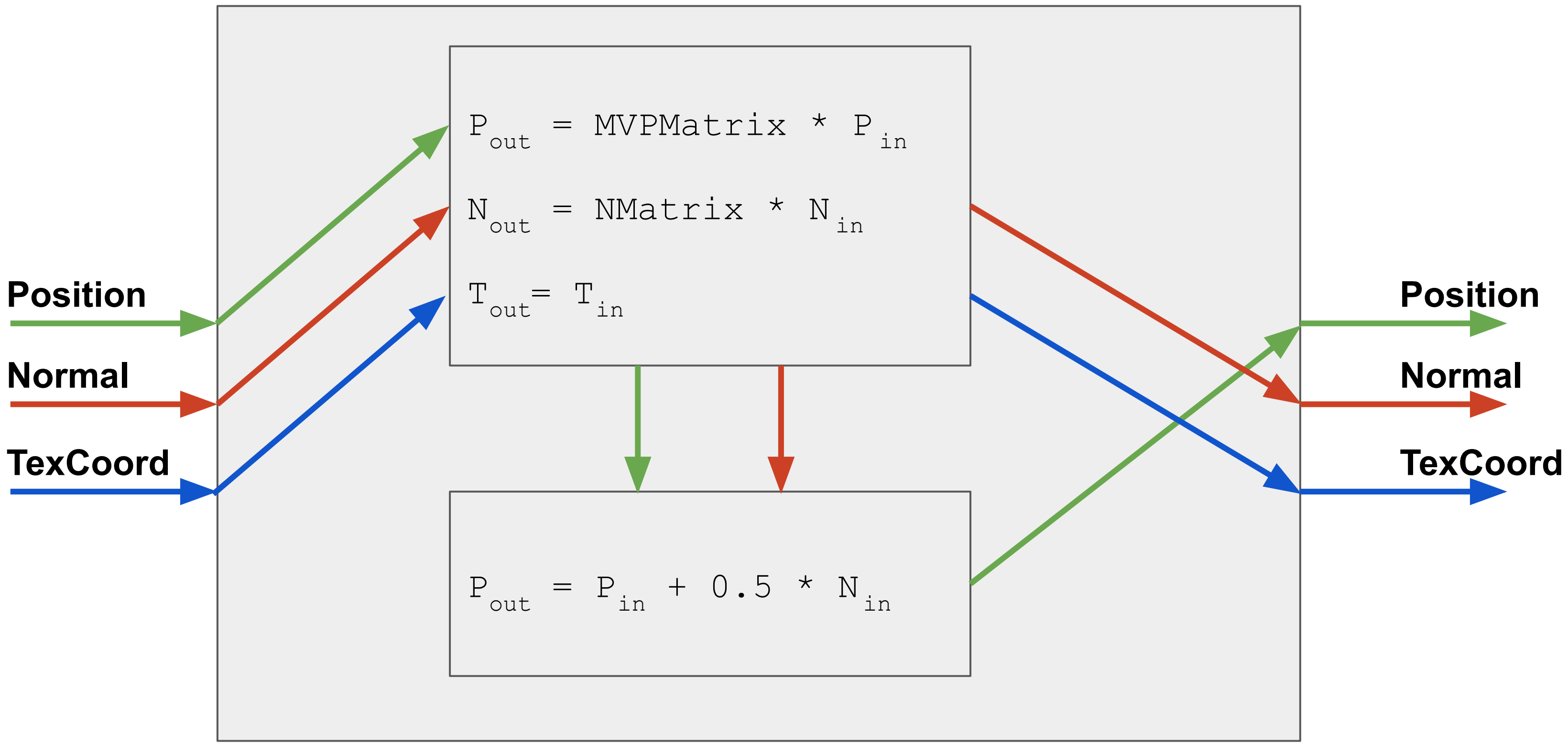
```
out vec4 Color;
void Fragment()
{
    Color = vec4(1,1,1,1);
}
```

Normals disappeared!

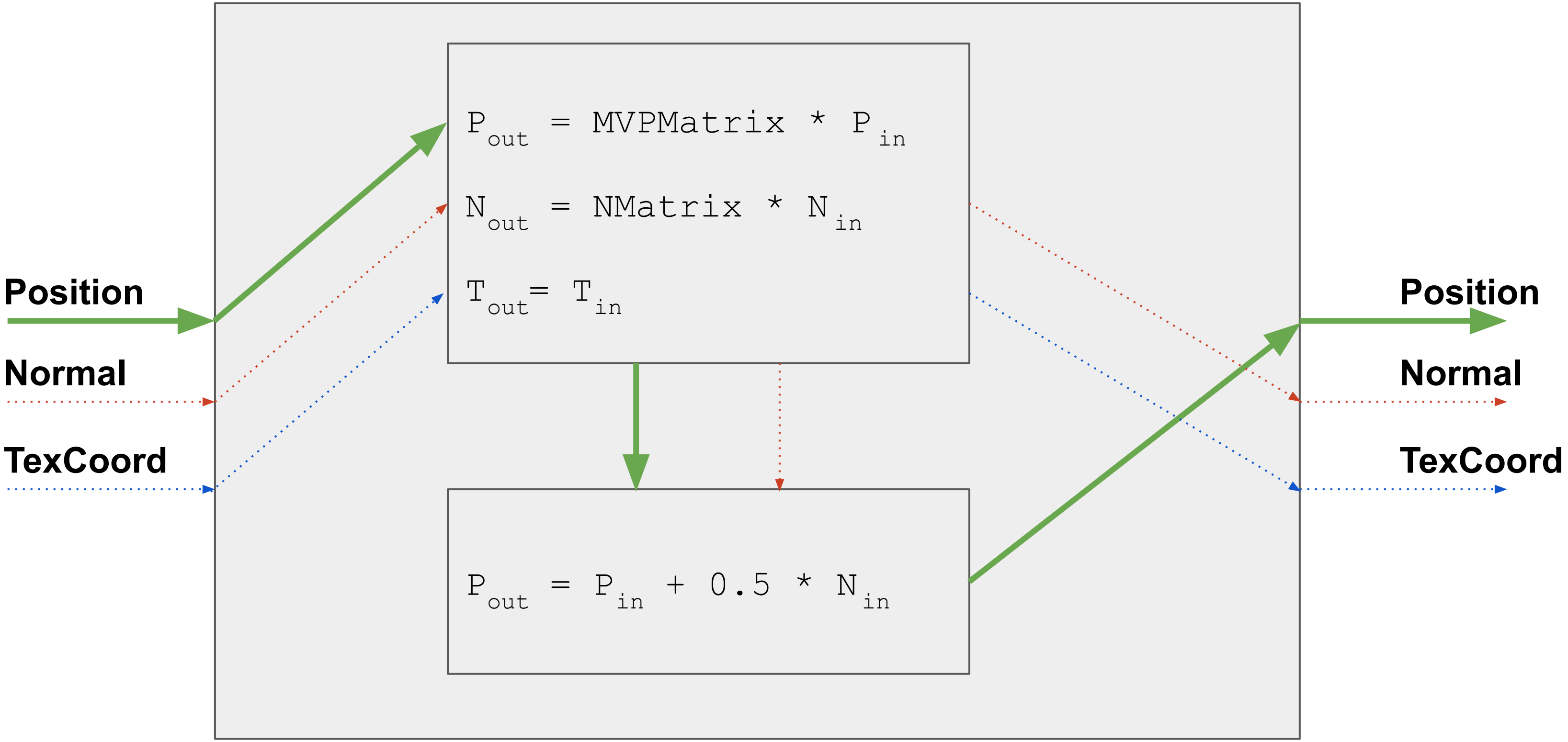
Shaders as Modules



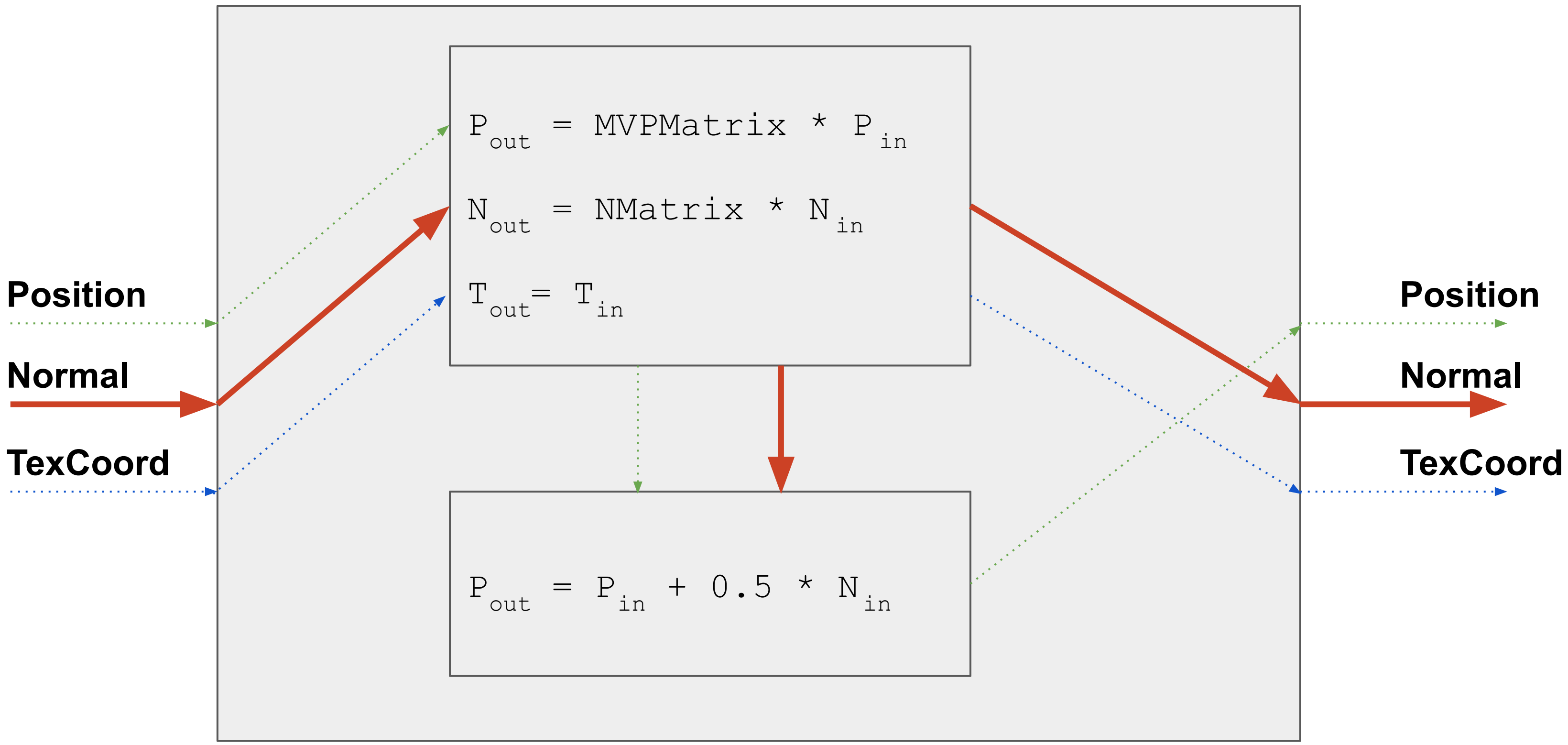
Shader Composition



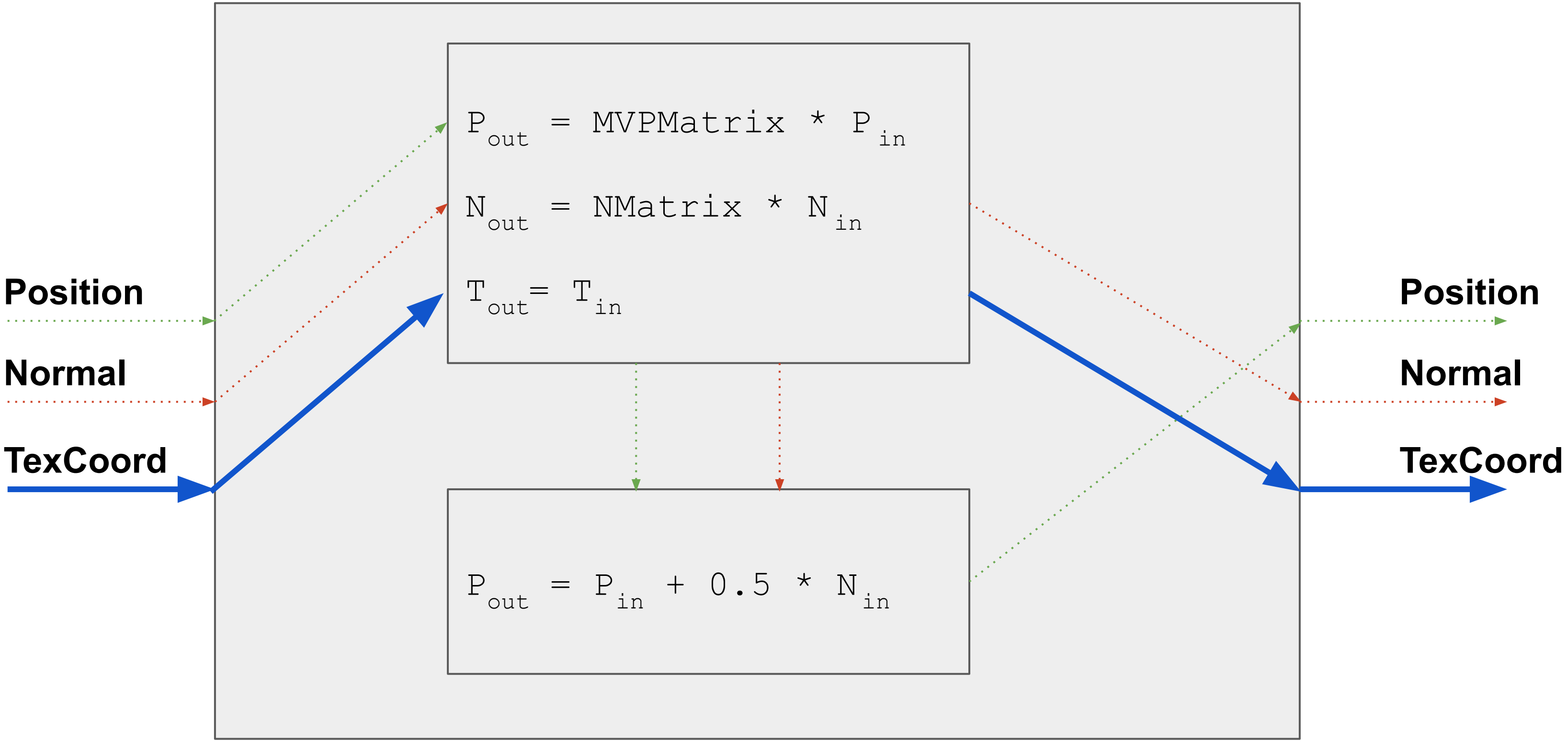
Shader Composition



Shader Composition



Shader Composition



Texturing: FShade

```
type Vertex =
    {
        [<TexCoord>]    tc       : V2d
        [<Color>]       color    : V4d
    }

let tex =
    sampler2d {
        addressU WrapMode.Wrap
        addressV WrapMode.Wrap
    }

let texturing (v : Vertex) =
    fragment {
        return { v with color = tex.Sample(v.tc) }
    }
```

Texturing: GLSL output

```
compose [ transform; white; texturing ]
```

```
uniform UBO{
    mat4 MVPMat;
};

in vec4 Position;
in vec2 TexCoord;
out vec2 fs_TexCoord
void Vertex()
{
    vec4 pos = MVPMat* Position;
    fs_TexCoord = TexCoord;
    gl_Position = pos;
}
```

```
uniform sampler2D tex;

in vec2 fs_TexCoord;
out vec4 Color;
void Fragment()
{
    Color = texture(tex, fs_TexCoord);
}
```

Implementation Overview

Implemented using F# quotations

Quotations allow us to get the function’s code as typed AST

Composition achieved via transforming/combining the AST

Translate AST to GLSL/HLSL/SpirV/etc.

```
let quotedAddition = <@ 1 + 3 @>
```

⇨

```
val quotedAddition : Quotations.Expr<int> =
  Call (None, op_Addition, [Value (1), Value (3)])
```

```
let rec spower (n : int) : Expr<int> -> Expr<int> =
        if n = 0 then fun _ -> <@ 1 @>
        elif n = 1 then fun t -> <@ %t @>
        else fun x -> <@ %x * (% spower (n-1) x) @>

spower 3 <@ 20 @>;;
```

⇨

```
val it : Quotations.Expr<int> =
 Call (None, op_Multiply,
    [Value (20), Call (None, op_Multiply, [Value (20), Value (20)])])
```

Optimizations

Dead Code Elimination

Constant Folding (-> evaluate constant expressions at compile time)

Function Inlining (-> reduce number of function calls)

Stage Hoisting (-> moving expressions out of loops if possible)

Common Subexpression Elimination (by using caches)

Many more...

Demos

DSLs in other domains

- Spiral, DSL for Digital signal processing, http://www.spiral.net/
  - Uses mathematical rules in order to generate variants of algorithms
  - Compiler automatically finds best FFT implementation
- Parallel programming: **Delite** by stanford PPL group, summer schoool talk
  - Also for heterogeneous parallel computing, e.g. Lee et al. 2011
  - OptiML: for machine learning
  - OptiQL: for data querying
  - OptiGraph: graph analytics
- LINQ
  - Originally for data querying
  - Interesting: LINQ provides higher order Functions, which SQL (target) lacks. This is why: Embedding by Normalization

references

[1] He, Yong, et al. "Shader components: modular and high performance shader development." *ACM Transactions on Graphics (TOG)* 36.4 (2017): 1-11. http://graphics.cs.cmu.edu/projects/shadercomp/he17_shadercomp.pdf

[2] McCool, Michael, and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. CRC Press, 2009 https://redirect.cs.umbc.edu/~olano/s2005c37/ch07.pdf

[3] Seitz Jr, Kerry A., et al. "Staged metaprogramming for shader system development." *ACM Transactions on Graphics (TOG)*38.6 (2019): 1-15. https://seitz.tech/publications/2019/slides/Selos-SIGGRAPHAsia2019.pdf

[4] FOLEY, TIM. "Slang: language mechanisms for extensible real-time shading systems." (2018). http://graphics.cs.cmu.edu/projects/slang/he18_slang.pdf

[5] Foley, Tim, and Pat Hanrahan. "Spark: modular, composable shaders for graphics hardware." *ACM Transactions on Graphics (TOG)* 30.4 (2011): 1-12. http://graphics.stanford.edu/papers/spark/spark_preprint.pdf

[6] Metashade https://github.com/ppenenko/metashade

[7] TFX, Bungie, 2017 https://advances.realtimerendering.com/destiny/gdc_2017/index.html#:~:text=The%20TFX%20language%20presents%20a,code%2C%20and%20related%20GPU%20states.

[8] Open Shading Language https://github.com/AcademySoftwareFoundation/OpenShadingLanguage

[9] Haaser, Georg, et al. "Cosmo: Intent-based composition of shader modules." *2014 International Conference on Computer Graphics Theory and Applications (GRAPP)*. IEEE, 2014. https://www.cg.tuwien.ac.at/sites/default/files/course/4041/attachments/RendEng-2015-12-14-paper.pdf

[10] https://developer.nvidia.com/vulkan-shader-resource-binding

[11] Abstract Shade Trees https://casual-effects.com/research/McGuire2006ShadeTrees/index.html

[12] Rautek et al. 2014, https://www.cg.tuwien.ac.at/research/publications/2014/Rautek_Peter_2014_VSA/Rautek_Peter_2014_VSA-Paper.pdf