Domain Specific Languages for Rendering Engines
Motivation: Programmable Shaders

● Texture combiners (for NV Riva TNT, 1999 :))
● Programmable shaders (ARB_vertex_program,..)
● High level shaders Cg [Mark et al.2003], HLSL [Microsoft 2012], GLSL [Kessenich et al. 2012]

```c
vec4 texel = texture2D(Texture0, TexCoord0);  
texel *= texture2D(Texture1, TexCoord1);  
gl_FragColor = texel;
```

```c
glBindTexture(GL_TEXTURE_2D, textureID1);  
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);  
//Sample RGB, multiply by previous texunit result

glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);  
//Sample ALPHA, multiply by previous texunit result

glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA, GL_PREVIOUS);  
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_ALPHA, GL_TEXTURE);  
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA, GL_SRC_ALPHA);  
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_ALPHA, GL_SRC_ALPHA);  
```
Motivation: Shader Trees

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

Many tried this, but not that practical...
Motivation

● DSL = language tailored to a specific use case
  ○ Makefiles, CSS, HTML, GLSL, RenderMan
● Provide **rendering engine** with API for working with shaders
● Provide possibility to write shaders directly in a **DSL**
  ○ Killer feature of rendering engine
  ○ Which is then mapped to API specific code such as GLSL
● In this first part of the lecture we will look at **DSL related work**
● In the second part we will look at concrete **shader languages and implementation techniques**
● Techniques apply to shaders but have other use cases (also in engines!)
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

What about:
  - **Animation/Storytelling**: when designing animations, we don’t care about classes.
  - **Shaders**: Shaders are simple best expressed as expression tree, where expression trees can easily be written as code.
  - **Scene description**, e.g. RenderMan uses own language to model scenes

Two reasons to implement own language:
  - Whenever, a language is a **better description** of software engineering speek
  - Whenever, we would like to do **optimizations** based on the domain-specific semantics
Overview

- Two types
  - External Domain Specific Language
  - Embedded (Internal) Domain Specific Language (EDSL)

- External DSL:
  - Use lexer, parser, program analysis and code generation for specific problem
  - Basically what you do in compiler construction courses
  - However, there are many tools helping here (e.g. YACC, spoofax)
  - Advantage: freedom in language design

- Embedded DSL:
  - Try to use host language to express domain specific constructs
    - e.g. by using implicit conversions, higher order functions etc
  - Advantages
    - fluently embedded in host program
    - can interact with embedding environment
Embedded Domain Specific Languages

- Two implementation techniques
  - 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]

```csharp
var transformed = input.DoByVertex(v => {
    v.WorldPosition = Uniform.ModelTrafo * v.Position;
    return v;
});
```
Implementation of Embedded DSLs

- **Shallow embedding**
  - Host language operations immediate translate to target language or interpretation
  - Optimizations are hard

- **Deep embedding**
  - Host language builds an expression tree
  - Expression tree can be analyzed and optimized
  - In order to be used for
    - Code generation
    - Interpretation
Embedded Expression Trees for Code Generation

- Represent syntax of DSL as classes
- Direct mapping of expression tree
- Each operation is a new node type
- Each DSL element has member for generating code (or via visitors et al)
- Resulting code has tree form
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))

Eliminating duplicated code is hard

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

Implicit conversion
Operator overload
Representing Expressions as Graphs

- Each code definition generates a symbol
- Before generating a symbol the EDSL syntactical object is looked for in symbol table
- This reduces code bloat
- Code is effectively stored as graph instead of flat tree.

```java
public class Lit2 : Exp {
    int value;
    public Lit2(int l) { value = l; }
    public override string Compile() {
        // either lookup definition for expression or, introduce
        // fresh symbol and return its usage.
        return CreateSymbol(this, () =>
            string.Format("{0}", value)
        );
    }
}
```

```
x_0 = 2
x_1 = 5
x_2 = (x_0 + x_1)
x_3 = (x_2 + x_2)
gls1: x_3
```
Functional Reactive Animation

FRAN, [Elliott 1997]

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

- **Embedded: Vertigo** [Elliot 2004], Embedded DSL in Haskell, Code generation generates Vertex/Fragment shader (assembly)

```
cylinder h = onZ (h·) ◦ revolve (λy → (1, y))
eggcrateCylinder h fm = displace (cylinder h) (freqMag eggcrate fm)

displace :: Surf → HeightField → Surf
displace surf field = surf + field · normal surf

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

- **External DSL: RenderMan**
DSL: Abstraction without regret

- 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

```scala
def fft(xs: Array[Complex]): Array[Complex] = xs match {
  case (x :: Nil) => xs
  case _ =>
    val N = xs.length // assume it's a power of two
    val (even0, odd0) = splitEvenOdd(xs)
    val (even1, odd1) = (fft(even0), fft(odd0))
    val (even2, odd2) = (even1 zip odd1 zipWithIndex) map {
      case ((x, y), k) =>
        val z = omega(k, N) * y
        (x + z, x - z)
    }.unzip;
    even2 :: odd2
}
```

This code gets compiled into numeric High-performance code, all inlined!

Lightweight Modular Staging, Rompf et al. 2010
DSLs in other domains

- **Spiral**, DSL for Digital signal processing, [http://www.spiral.net/](http://www.spiral.net/)
  - Uses mathematical rules in order to generate variants of algorithms
  - Compiler automatically finds best FFT implementation

- **Parallel programming**: [Delite](http://www.spiral.net/) by Stanford PPL group, *summer school talk*
  - Also for heterogeneous parallel computing, e.g. [Lee et al. 2011](http://www.spiral.net/)
  - 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](http://www.spiral.net/)
Embedding by quoting

- Some languages support quoting
  - LISP
  - OCaml, F#
- Helps in building DSL [Syme 2006]
- Recent work on CPU-GPU staging [Sampson et al. 2017]

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

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

- DSLs are useful for in two scenarios.
  - Rendering engine modules exposed to designers/artists etc. should use adequate language
  - Domain specific compilers can do domain specific optimizations
- There is **HUGE** set of related work in other domains
- DSLs recently became hot in Visualization as well (e.g. Vega)
- Two implementation techniques
  - External DSL
  - Embedded DSL
- For embedded DSL watch out for duplicated expressions
- Typically, you need many tricks like implicits etc
Further Reading (1)

- **Summer school videos on domain specific languages** [https://www.youtube.com/channel/UCfrkueNadcqDv-YXFmn_CdA](https://www.youtube.com/channel/UCfrkueNadcqDv-YXFmn_CdA)
- An image synthesizer, Perlin 1985, [https://dl.acm.org/citation.cfm?id=325247](https://dl.acm.org/citation.cfm?id=325247)
- Vega, a Visualization Grammar, [https://vega.github.io/vega/](https://vega.github.io/vega/)
Further Reading (2)

- Spiral, http://www.spiral.net/
Further reading (3)

- Static stages for heterogeneous programming, Sampson 2017, https://dl.acm.org/citation.cfm?id=3133895