Scene Representation for dynamic scenes
Collecting renderable things
Recap: Use render objects as intermediate repr.

- Use notion of renderables (in the following referred to as render object)
- Graphics code or scene graph implementation assembles renderables
- Renderable objects interpreted by render loop
- Render object contains all graphics state

```python
list { RenderObject_1,
    RenderObject_2,
    ... }  
```

```python
for ro in renderObjects:
    Graphics.setViewTrafo(ro.Trafo
    Graphics.setShader ro.Shader
    Graphics.render ro.Geometry
```
Render objects revisited...

- Render objects consist of:
  - Rasterizer state
  - BlendState
  - Viewport
  - Shader
  - Uniform values
  - Vertexbuffers*
  - Indexbuffer
  - Instancebuffer*
  - Draw call description
  - ....
type RenderObject = {
    DrawCallInfos : list<DrawCallInfo>
    IndirectBuffer : IIndirectBuffer
    Mode : IndexedGeometryMode
    Surface : Surface
    DepthTest : DepthTestMode
    CullMode : CullMode
    BlendMode : BlendMode
    FillMode : FillMode
    StencilMode : StencilMode
    Indices : Option<BufferView>
    InstanceAttributes : IAttributeProvider
    VertexAttributes : IAttributeProvider
    Uniforms : IUniformProvider
    ConservativeRaster : bool
    Multisample : bool
    WriteBuffers : Option<Set<Symbol>>
}

typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {
    ID3D12RootSignature *pRootSignature;
    D3D12_PRIMITIVE_TOPOLOGY_TYPE PrimitiveTopologyType;
    D3D12_SHADER_BYTECODE VS;
    D3D12_SHADER_BYTECODE PS;
    D3D12_SHADER_BYTECODE DS;
    D3D12_SHADER_BYTECODE HS;
    D3D12_SHADER_BYTECODE GS;
    D3D12_STREAM_OUTPUT_DESC StreamOutput;
    D3D12_BLEND_DESC BlendState;
    UINT SampleMask;
    D3D12_RASTERIZER_DESC RasterizerState;
    D3D12_DEPTH_STENCIL_DESC DepthStencilState;
    D3D12_INPUT_LAYOUT_DESC InputLayout;
    D3D12_INDEX_BUFFER_STRIP_CUT_VALUE IBStripCutValue;
    UINT NumRenderTargets;
    DXGI_FORMAT RTVFormats[8];
    DXGI_FORMAT DSVFormat;
    DXGI_SAMPLE_DESC SampleDesc;
    UINT NodeMask;
    D3D12_CACHED_PIPELINE_STATE CachedPSO;
    D3D12_PIPELINE_STATE_FLAGS Flags;
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;
Approach 1:
Traverse scene data (each frame) and emit graphics commands

Pros:
- easy to in old school OpenGL

Cons:
- harder in stateless APIs, more state tracking at engine level, inefficient
Approach 2:
Traverse scene data and record graphics commands

Pros:
- Easy to implement in D3D12, Vulkan

Cons:
- Inefficient, since command lists need to be rebuilt repeatedly
Approach 2.5:
Traverse scene data and emit graphics commands

Pros:
- Easy to implement in D3D12, Vulkan

Cons:
- Hard to implement in OpenGL
Unreal Engine architecture

- Original architecture
  - Game Thread enqueues rendering commands
  - Rendering Thread generates Vulkan Cmd Buffers

Unreal Engine architecture (improved)

- Render Hardware Interface (RHI)
  - Cross platform way to talk to each Gfx API
- Improved architecture
  - Game Thread enqueues rendering commands
  - Rendering Thread generates RHI command list
  - RHI Thread translates into Vulkan Cmd Buffers
Unreal Engine architecture (multithreaded)

- UE4 RHI Architecture is multithreaded
  - N Render threads with M RHI Threads

Idea 3: incremental evaluation (approach we present here)

Virtual machine traverses RenderObject list:

```
for ro in renderObjects:
    Graphics.setViewTrafo ro.Trafo
    Graphics.setShader ro.Shader
    Graphics.render ro.Geometry
```

Virtual machine executes commands:

```
for drawCommand in commands:
    drawCommand()  # compute Δ & update array of graphics commands
```

[Haaser et al. 2014]
Big Picture

Step 1: Extract Render objects
{Ro₁, Ro₂, Ro₃} +Ro₄

Step 2: Prepare resources
{PRo₁, PRo₂, PRo₃} +PRo₄

Step 3: Optimize for GPU

Ro = render object
PRo = prepared render object (contains already GPU resources)
type PreparedRenderObject = {
    Program : IResource<Program, int>
    UniformBuffers : Map<int, IResource<UniformBufferView, int>>
    Uniforms : Map<int, IResource<UniformLocation, nativeInt>>
    Textures : Map<int, IResource<Texture, V2i> * IResource<Sampler, int>>
    Buffers : list<int * BufferView * AttributeFrequency * IResource<Buffer, int>>
    IndexBuffer : Option<OpenGl.Enums.IndexType * IResource<Buffer, int>>
    BeginMode : IResource<GLBeginMode, GLBeginMode>
    DrawCallInfos : IResource<DrawCallInfoList, DrawCallInfoList>
    IndirectBuffer : Option<IResource<IndirectBuffer, V2i>>
    VertexInputBinding : IResource<VertexInputBindingHandle, int>
    DepthTestMode : IResource<DepthTestInfo, DepthTestInfo>
    ...}
    StencilMode : IResource<GLStencilMode, GLStencilMode>
    Multisample : IResource<bool, int>
    ConservativeRaster : IResource<bool, int>

    ColorAttachmentCount : int
    DrawBuffers : Option<DrawBufferConfig>
    ColorBufferMasks : Option<list<V4i>>
    DepthBufferMask : bool
    StencilBufferMask : bool
}

Possible definition of (OpenGL) prepared render objects:
● Same content as render object
● But allocated resources and native GL specific states
type PreparedRenderObject =
{
    device : Device
    original : RenderObject

    resources : list<IResourceLocation>

    pipelineLayout : PipelineLayout
    pipeline : INativeResourceLocation<VkPipeline>
    indexBuffer : Option<INativeResourceLocation<IndexBufferBinding>>
    descriptorSets : INativeResourceLocation<DescriptorSetBinding>
    vertexBuffers : INativeResourceLocation<VertexBufferBinding>
    drawCalls : INativeResourceLocation<DrawCall>
    isActive : INativeResourceLocation<int>
    activation : IDisposable
}
Lazy Incremental Computation for Efficient Scene Graph Rendering - Revisited

- Wörister et al. 2013 already use render object notation (they called it render job)
- Scene graph caches collect render objects with special traversal
- Render caches can be updated
  - For value changes: incremental update
  - For structural changes: rebuild of complete cache
Dependency graphs

- First algorithms for evaluating programs incrementally
  - Standard optimal algorithm by Reps et al. 1983
  - Demand driven approach by Hudson 1991
- Lazy Incremental Computation for Efficient Scene Graph Rendering used Hudson’s eager marking, lazy evaluation algorithm

Render caches and their dependency graph in Wörister’s work.
Dependency graphs for arbitrary changes?

- Wörister’s notes + future work:
  - Since there is no notion of change, we cannot implement structural changes easily
  - The input graph is too complex to be updated incrementally
  - Input scene graph can be changed structurally, and we have no efficient way to detect this.

- Clearly, there is a need to talk about dynamism

- There are different implementation techniques
  - If we know all dependencies, we can write special code
    - For each modification in the scene, we implement efficient update mechanisms
  - Publish/Subscribe - subscribers update gpu resources
    - How about structural changes, how to unsubscribe nested subscriptions
  - Reactive programming Domain Specific Languages
General purpose incremental evaluation

- Incremental computation has a long history [Ramalingam and Reps `93]
- First appeared as special algorithms for implementing incremental attribute grammar systems
  - use case: interactive program editing environment, ‘Intellisense’ [Reps 1982]
  - Later was extended to graph algorithms, [Hoover 1986]
- Later many more special purpose algorithms invented (often called online algorithm)
  - Run algorithm once -> change input -> adapt output
  - Example: incremental convex hull

<table>
<thead>
<tr>
<th>Input</th>
<th>Algorithm</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changes $\Delta$  \rightarrow Change propagation  \rightarrow Minimal set of changes
Adaptive Functional Programming

- Later a general purpose incremental language was proposed
- Idea:
  - While running the program for the first time, track dependencies
  - If something changes in the input, perform change propagation, which makes the output consistent again.
- Interesting result:
  - No special compiler needed, can be implemented as library
- Acar’s implementation in ML
- Other implementations in Haskell
Towards adaptive programs

Publish/Subscribe:

```csharp
input.Subscribe(x =>
    x * 2
)
```

Basically says: on changed...

Computation, e.g. `input.Value * 2`
Towards adaptive programs

Computation, e.g. `inputA.Value + inputB.Value`

Publish/Subscribe:

```csharp
inputA.Merge(inputB).Subscribe(both =>
    both.Item1 + both.Item2)
```

Semantics:

```
s1 --1--1--1--1
s2 ---2---2---2
r   --12-1-21--2
```
Typical problems

- Publish/Subscribe/Rx based systems are push based
  - No batch updates possible (change inputA and inputB simultaneously and compute outputs only once)
  - Nested computations (dynamic values in dynamic values)

Merge semantics:

\[
\begin{align*}
  s1 & \rightarrow 1 \rightarrow 1 \rightarrow 1 \\
  s2 & \rightarrow 2 \rightarrow 2 \rightarrow 2 \\
  r   & \rightarrow 12 \rightarrow 1 \rightarrow 21 \rightarrow 2 \\
\end{align*}
\]

Dependency only exists if object is visible
Approach: Explicit notation of dependency graph (instead of callbacks)

- Use explicit notation of dependency graph
- Use explicit algorithm which performs change propagation
- This allows us to:
  - Provide batch changes
  - Change evaluation strategy in favor of rendering
    - **on-demand/pull based** evaluation as in Lazy Incremental Computation for Efficient Scene Graph Rendering)
- Domain Specific Language helps in working with adaptive programs (if they were static)
Dependency Graph Operations

- Creates modifiable input cell

```plaintext
let modRef1 = Mod.init 10
```

- Create single edge dependency

```plaintext
let mappedref = Mod.map (fun s -> s + 1) modRef1
```

- Mod.force evaluates a dependency graph

```plaintext
Mod.force mappedref ⇒ 11
```
Dependency Graph Operations

- Mod.change changes single input mod. Only valid in transact block. Transact block allows to do batch changes. When transaction returns, all outputs know that they need to be recomputed (eager marking).

```javascript
transact (fun () ->
    Mod.change modRef1 0
)
```

- Subsequent evaluations return consistent values (lazy evaluation)

```javascript
Mod.force mappedref ⇒ 1
```
Basic operations can be hidden beneath DSL

```
let a =
  adaptive {
    let! m = Mod.init 10
    let! c = Mod.init 20
    let! d = Mod.init 30
    return m + c + d
  }
```

Approach: Monads for incremental computing [Carlsson 2002]
Dependency Graphs may be dynamic

```ocaml
let c1 = adaptive {
  let! currentA = a
  return fib currentA
}

let c2 = adaptive {
  let! currentA = a
  return fac currentA
}

let d (d : aref<bool>) = adaptive {
  let! fibOrFac = d
  if fibOrFac then
    let! f = c1
    return f
  else
    let! f = c2
    return f
}
```
Evaluation Strategies

- Two approaches for evaluating dependency graphs.
- Acar et al. 2002 use eager evaluation
  - conceptually, transact block recomputes all outputs directly but batched
- Hammer et al. 2014 introduced incremental language which
  - uses out of date marking (similar to Hudson and Wörister)
  - And evaluates values in an on demand-manner
Incremental evaluation for computing render objects

- Two choices
  - Extract render objects from game/application state - make it fast, use multithreading
  - Extract render objects once and update them using incremental techniques
Programming Aardvark - A Tutorial
Recap & Takeaways

● Current status of the lecture
  ○ Discussed mechanisms for representing scenes
  ○ Talked about handling value changes efficiently using dependency graphs
  ○ If we want fully dynamic scenes
    ■ Re-traverse the game/application state each frame in order to build render commands.
    ■ Either we need to implement each modification and its reaction specifically,
    ■ ..or we need a mechanism for expressing dynamism generically
  ○ There are various algorithms for incremental computation
    ■ Lazy Incremental Computation for Efficient Scene Graph Rendering uses eager marking and lazy evaluation, but for resource updates they use dependency graph polling.
    ■ Acar’s approach [e.g. Acar et al. 2002] provide theoretical framework for incremental evaluation
    ■ Hammer’s approach [Hammer et al. 2014] uses eager marking and lazy evaluation which seems suited for rendering (e.g. because of culling)
Next steps

- We have talked about optimization techniques for GPUs
  - However, so far we did not look at completely dynamic scenes
- Next, lecture we will look at low level optimization techniques in the face of fully dynamic scenes.
- Next, we will look into shader programming and domain specific languages for shading

- At that point, all should be fine to look into material/light systems and practical algorithms/applications thereof.
Further reading

- A categorized bibliography on incremental computation, Ramalingam and Reps 1993, https://dl.acm.org/citation.cfm?id=158710
- Incremental graph evaluation (attribute grammar), Hoover 1987, https://dl.acm.org/citation.cfm?id=37566