Geometry shader

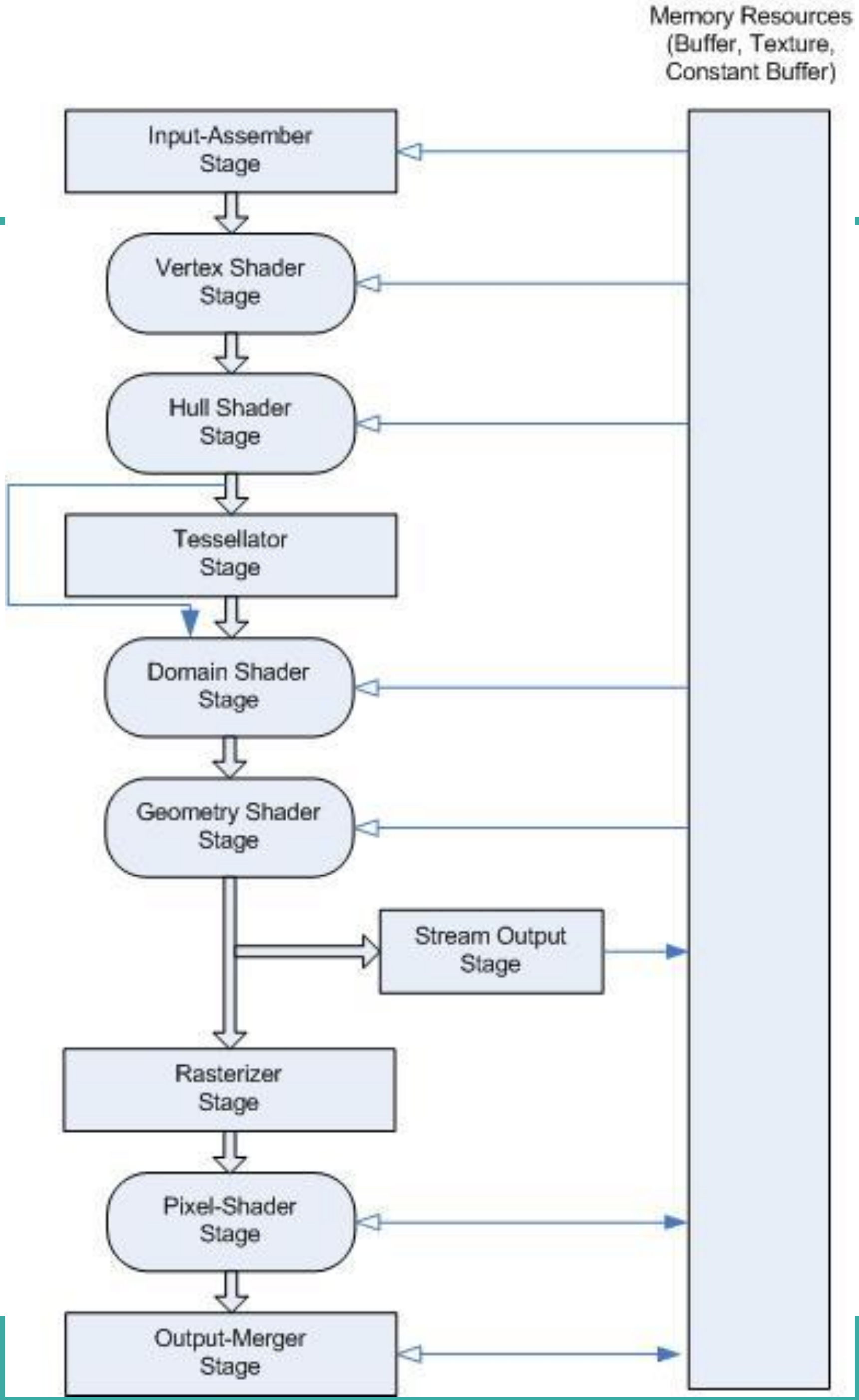
CMP301 Graphics Programming with Shaders

This week

- Geometry shader
  - What is it?
  - It’s place in the pipeline
  - Uses
  - Example

Geometry shader

- Occurs prior to rasterisation
- And after the tessellation stages



Geometry shader

- Receives entire primitives
  - Point
  - Line
  - Triangle
- Executed for every primitive
  - For example every triangle in a triangle list
- Output primitive does not need to match input primitive
  - E.g. Receive point, output triangle

Geometry shader

- Can create and destroy geometry
  - The vertex shader can’t do this
  - Expand primitives
  - Cull primitives based on user defined condition
- Output is defined by a vertex list
  - MUST be transformed into homogeneous clip space
  - Translation: apply world, view and proj matrix here

Geometry shader

- How many primitives can I output?
  - Lots
- Should I?
  - Probably not, depends how many
- The maximum number of output primitives allowed is based on the output structure
- For normal usage, we have to specify how many vertices the GS outputs per evocation
  - Peek performance is 1-20
  - Performance halves for 27-40 scalars

Geometry shader use

- Generating geometry
  - Given vertices, spawn cubes, voxels
  - Point sprite rendering
  - Including particles systems and billboard sprites
- Geometry subdivision
  - Given a triangle it is possible to create smaller triangles
  - However this is hugely inefficient and better done with tessellation
- Geometry manipulation
  - Manipulate whole polygons (not a single vertex)
  - Generate data; normals, texture coordinates, etc

Geometry shader

- Main function returns void
  - Unlike the other shader functions we have called so far
- Received two input parameters
  - Primitive data
  - Stream-output object

Geometry shader

- Example function calls

```
[maxvertexcount(4)]
void main(point InputType input[1], inout TriangleStream<OutputType>
triStream)
```

- Or

```
[maxvertexcount(3)]
void main(triangleadj InputType input[6], inout
TriangleStream<OutputType> triStream)
```

Geometry shader

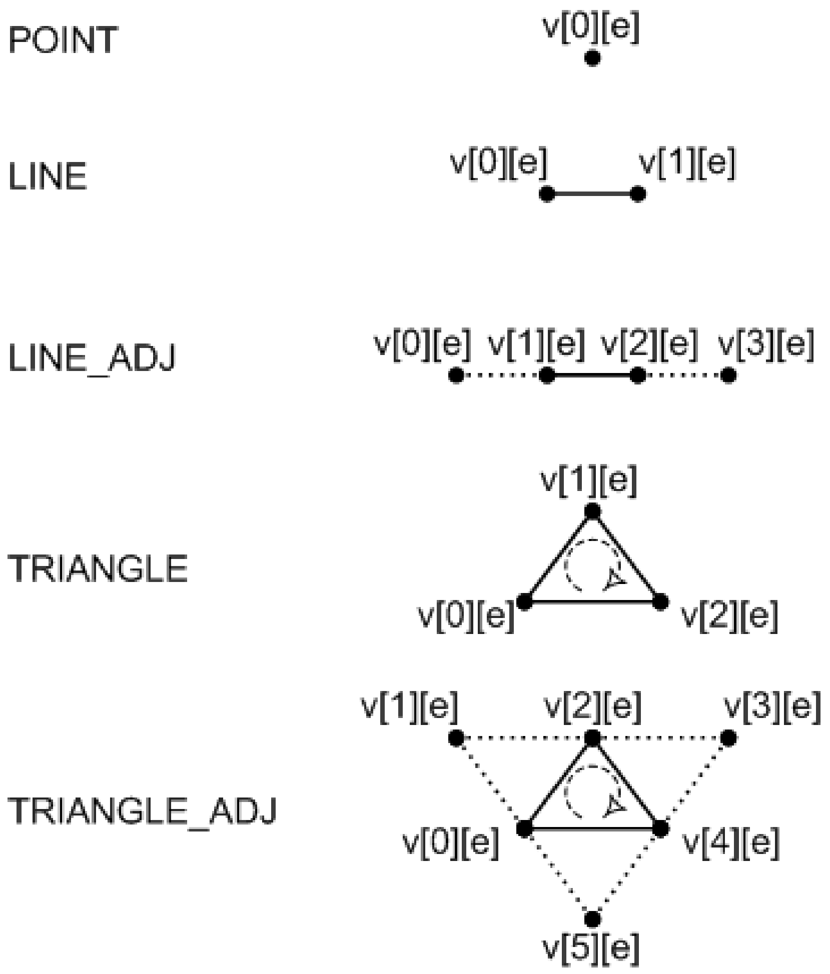
- Geometry shader specific parameters
  - Maxvertexcount(n)
  - Required
  - The number of vertices output by a single pass of the geometry shader
  - Specified prior to the geometry shader main function
    - Similar to the parameters setting in the hull and domain shaders
  - [maxvertexcount(3)]

Geometry shader

- Primitive data input
  - Primitive type
  - Data type
  - Name
  - [Number of elements]

Geometry shader

- Primitive type and number of elements
  - Point – 1
    - Operating on one point at a time
  - Line – 2
    - Line requires two vertices
  - Triangle – 3
  - Lineadj – 4
    - Two for the line, plus two for adjacent lines
  - Triangleadj – 6
    - Borders three additional triangles
- Adjacent does not work if tessellating



Geometry shader

- Data type
  - Struct defining the data being received
  - Vertex / domain output
    - Position, tex, normals, etc
- Name
  - Name of variable
- [Number of elements]
  - Array size of input
  - Depends on the primitive type

Geometry shader

- Example function calls

```
[maxvertexcount(4)]
void main(point InputType input[1], inout TriangleStream<OutputType>
triStream)
```

- Or

```
[maxvertexcount(3)]
void main(triangleadj InputType input[6], inout
TriangleStream<OutputType> triStream)
```

Geometry shader

- Stream-output object
  - Streams data out of the geometry shader
- Defined with
  - Stream-output object type
  - < Data type >
  - Name

Geometry shader

- Stream-output object type
  - Three types
  - PointStream
    - Sequence of point primitives
  - LineStream
    - Line primitives
  - TriangleStream
    - Triangle primitives
- < Data type >
  - Any data type
  - In most cases a struct defining data for the next shader

Geometry shader

- Example function calls

```
[maxvertexcount(4)]
void main(point InputType input[1], inout TriangleStream<OutputType>
triStream)
```

- Or

```
[maxvertexcount(3)]
void main(triangleadj InputType input[6], inout
TriangleStream<OutputType> triStream)
```

Geometry shader

- Steam-output object has two functions
  - Append()
    - Append output data to the stream
  - RestartStrip()
    - End the current primitive strip and start a new one

Geometry shader

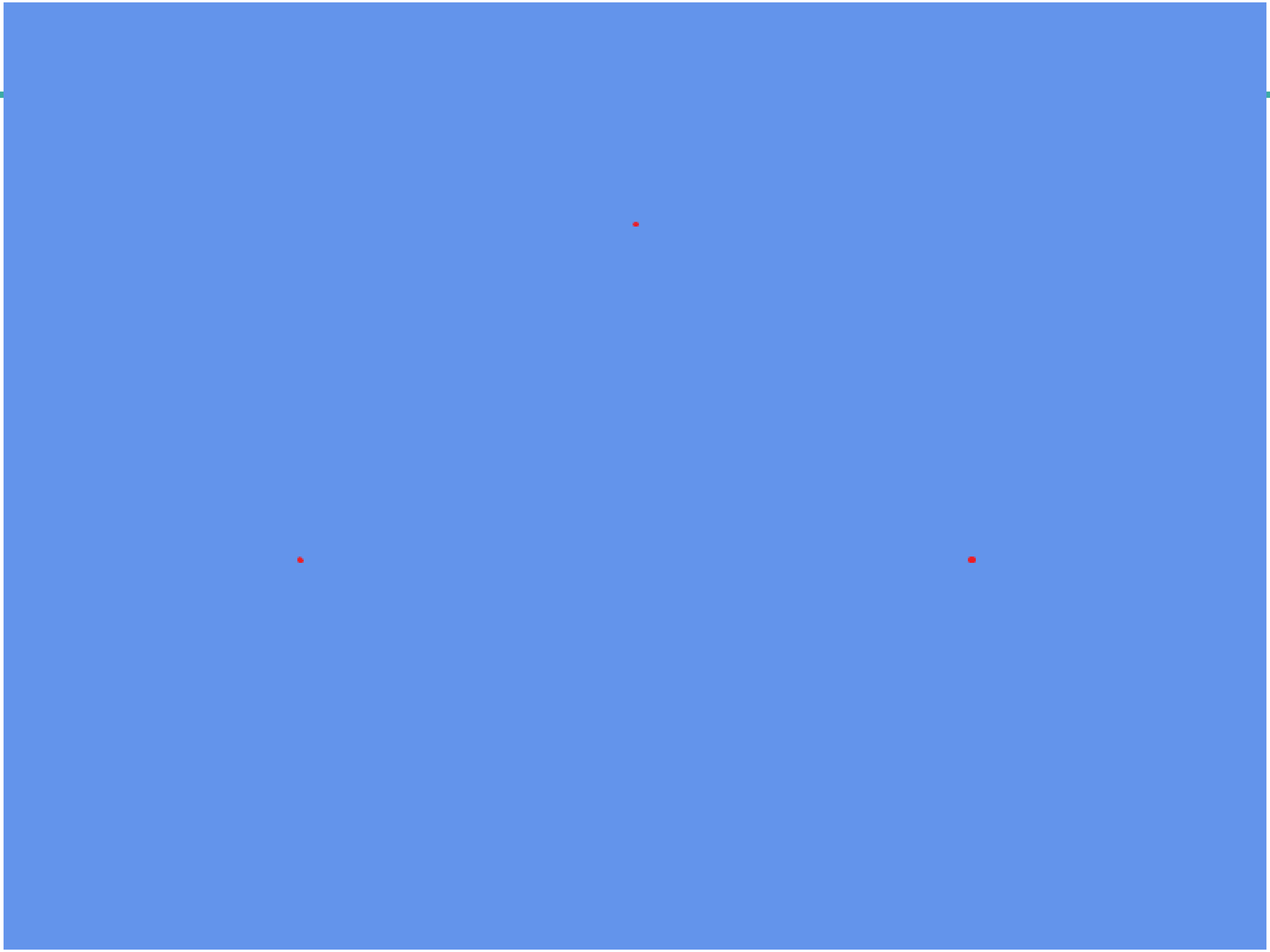
- PrimitiveID
  - Use of the semantic informs the AI stage to generate a primitive ID
  - Uint
  - Starts at 0
  - Give a unique ID for each primitive per draw call
- Can come in handy
  - Use it for tracking which primitive you are processing
  - Based on ID can swap between texture or processing

Example

- Simple example
  - Send point data
  - Render triangle for each point
- Required
  - Modified mesh class
  - Three shaders (vs, gs, ps)
  - Shader class

Modified mesh

- In the framework there is a point mesh class
- Different topology type
  - D3D11_PRIMITIVE_TOPOLOGY_POINTLIST
  - Not triangle list (but could be)
  - Not control point list, because no tessellation (but could be)
- It contains three points of a triangle but normal rendering wouldn’t draw a triangle, just three points



Example

- Given the simple triangle geometry point mesh
  - Use the geometry shader to generate a triangle at each point

triangle_vs.hlsl

```
struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};


InputType main(InputType input)
{
// No processing required so Vertex shader passes values onto next stage.
// You could manipulate the points in the mesh before passing them on.
return input;
}
```

Triangle_gs.hlsl

```hlsl
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

// pixel input type
struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Triangle_gs.hlsl

```
[maxvertexcount(3)]
void main(point InputType input[1], inout TriangleStream<OutputType> triStream)
{
    OutputType output;

    // Change the position vector to be 4 units for proper matrix calculations.
    input[0].position.w = 1.0f;

    // Move the vertex away from the point position
    output.position = input[0].position + float4(0.0, 1.0, 0.0, 0.0);

    output.position = mul(output.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    output.tex = input[0].tex;

    output.normal = mul(input[0].normal, (float3x3)worldMatrix);
    output.normal = normalize(output.normal);

    triStream.Append(output);
```

Triangle_gs.hlsl

```
output.position = input[0].position + float4(-1.0, 0.0, 0.0, 0.0);
output.position = mul(output.position, worldMatrix);
output.position = mul(output.position, viewMatrix);
output.position = mul(output.position, projectionMatrix);
output.tex = input[0].tex;
output.normal = mul(input[0].normal, (float3x3)worldMatrix);
output.normal = normalize(output.normal);

triStream.Append(output);

output.position = input[0].position + float4(1.0, 0.0, 0.0, 0.0);
output.position = mul(output.position, worldMatrix);
output.position = mul(output.position, viewMatrix);
output.position = mul(output.position, projectionMatrix);
output.tex = input[0].tex;
output.normal = mul(input[0].normal, (float3x3)worldMatrix);
output.normal = normalize(output.normal);

triStream.Append(output);

triStream.RestartStrip();

}
```

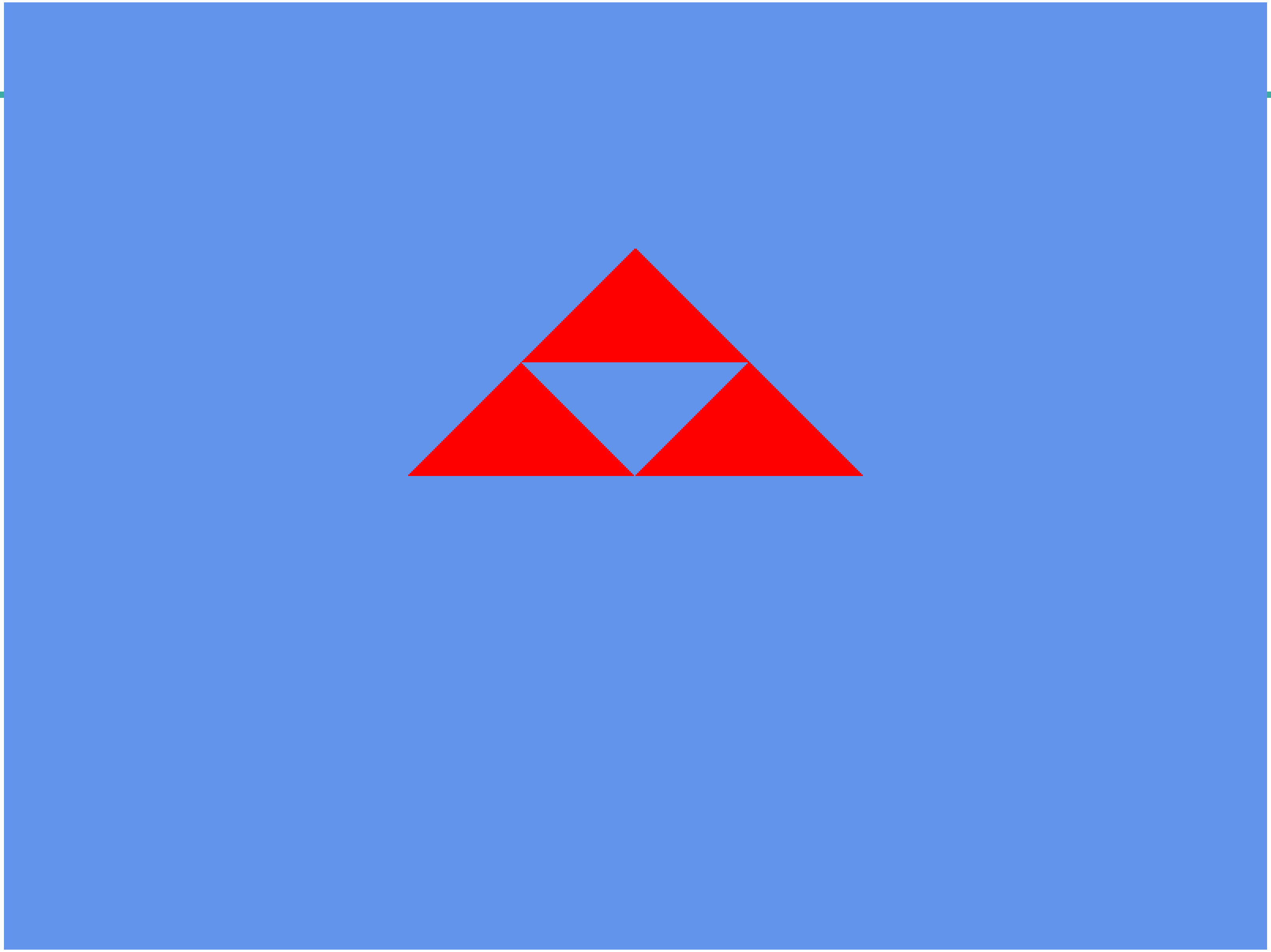
Pixel shader

```
Texture2D texture0 : register(t0);
SamplerState Sampler0 : register(s0);

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

float4 main(InputType input) : SV_TARGET
{

    return float4(1.0, 0.0, 0.0, 1.0);
}
```



More results

- Improve output generation
- Instead of hard coding EVERYTHING
- You can define or pass in a constant buffer with size data for each sprite
    - Data defining the four corners of a quad
    - There position is defined by the input vertex / point

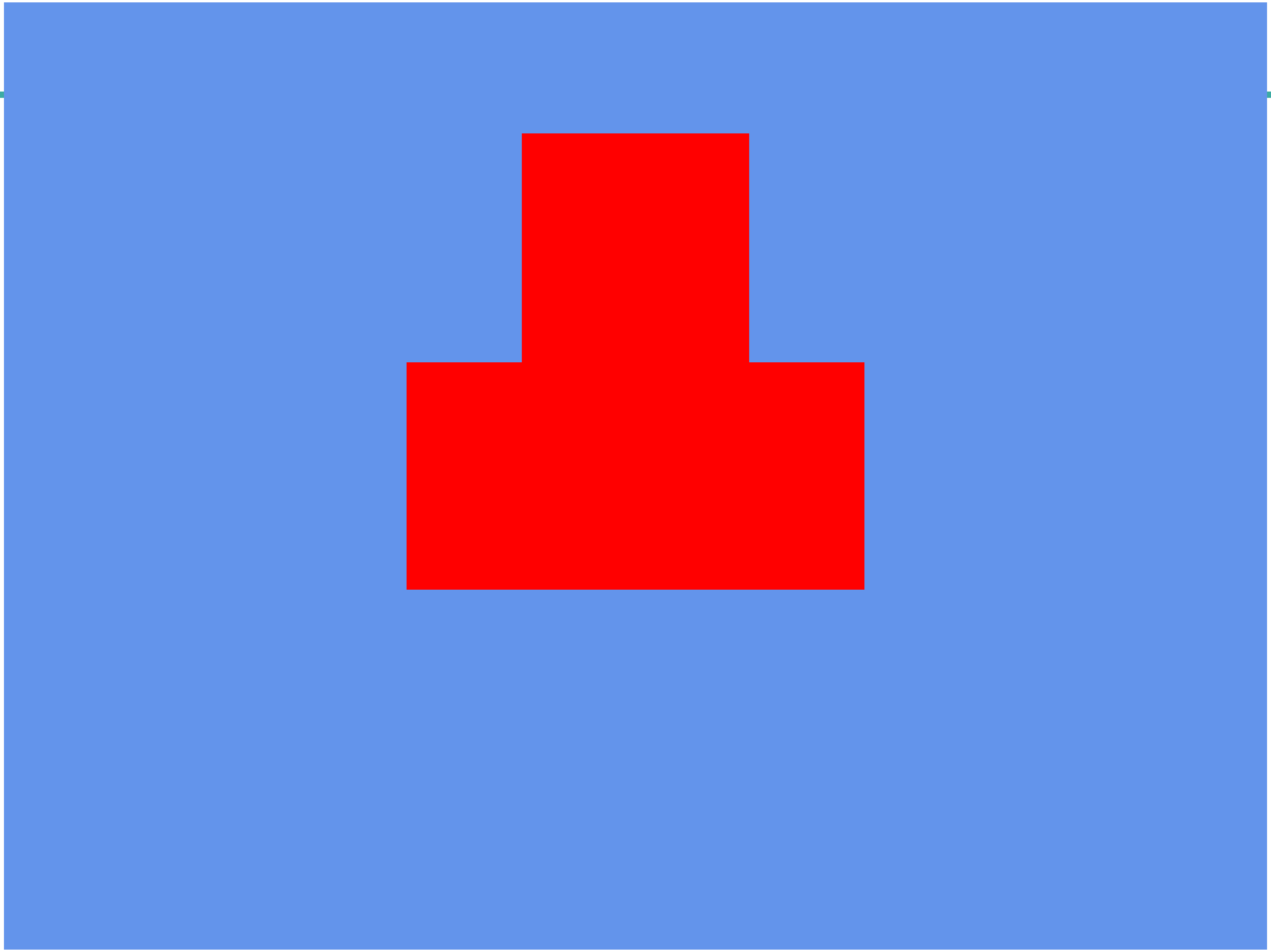
Quads!

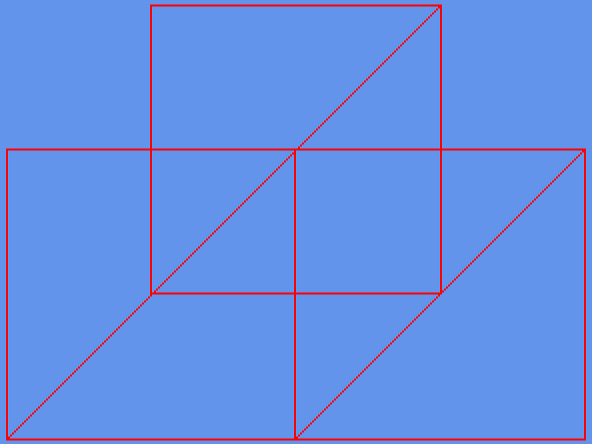
```
cbuffer PositionBuffer
{
 static float3 g_positions[4] =
    {
        float3( -1, 1, 0 ),
        float3( -1, -1, 0 ),
        float3( 1, 1, 0 ),
        float3( 1, -1, 0)
    };
};
```

Quads!

```
for(int i=0; i<4; i++)
{
        float3 vposition = g_positions[i];
        vposition = mul( vposition, (float3x3) worldMatrix ) + input[0].position;
        output.position = mul( float4(vposition,1.0), viewMatrix );
        output.position = mul(output.position, projectionMatrix);


        output.tex = input[0].tex;
        output.normal = input[0].normal;
        triStream.Append(output);
}
```





Uses of geometry shader

- Point sprite generation
    - Including billboarding
- Particle systems
- Shadow volume extrusion
- Rendering normals
- Destroyable models

In the labs

- The lab
  - Working with the geometry shader

- Next week
  - Revision plus GUI lecture
  - Give me revision topics!