DirectX 11
First Elements

0.

Project changes

changed Dx11Base.h

```
void Update( float dt );
void Render( );
```

to

```
virtual void Update( float dt ) = 0;
virtual void Render( ) = 0;
```

new App3D.h

```
#ifndef _APP_3D_H_
#define _APP_3D_H_

#include "Dx11Base.h"

class App3D : public Dx11Base
{
   public:
      App3D( );
      virtual ~App3D( );

      bool LoadContent( );
      void UnloadContent( );

      void Update( float dt );
      void Render( );
};

#endif
```

new App3D.cpp

```
#include"App3D.h"
#include<xnamath.h>

App3D::App3D() {}
App3D::~App3D() {}
void App3D::UnloadContent() {}
void App3D::Update(float dt) {}
bool App3D::LoadContent() { return true; }

void App3D::Render()
{
   if( d3dContext_ == 0 )
      return;

   float clearColor[4] = { 0.0f, 0.0f, 0.25f, 1.0f };
   d3dContext_->ClearRenderTargetView(
                          backBufferTarget_, clearColor );

   swapChain_->Present( 0, 0 );
}
```

removed from Dx11Base.cpp

```
void Dx11Base::Update( float dt )
{}

void Dx11Base::Render()
{
	if (d3dContext_ == 0)
		return;

	float clearColor[4] = { 0.0f, 0.0f, 0.25f, 1.0f };
	d3dContext_->ClearRenderTargetView(backBufferTarget_, clearColor);

	swapChain_->Present(0, 0);
}
```

changed main.cpp

#include “App3D.h”

…

Dx11Base app3D;

to

App3D app3D;

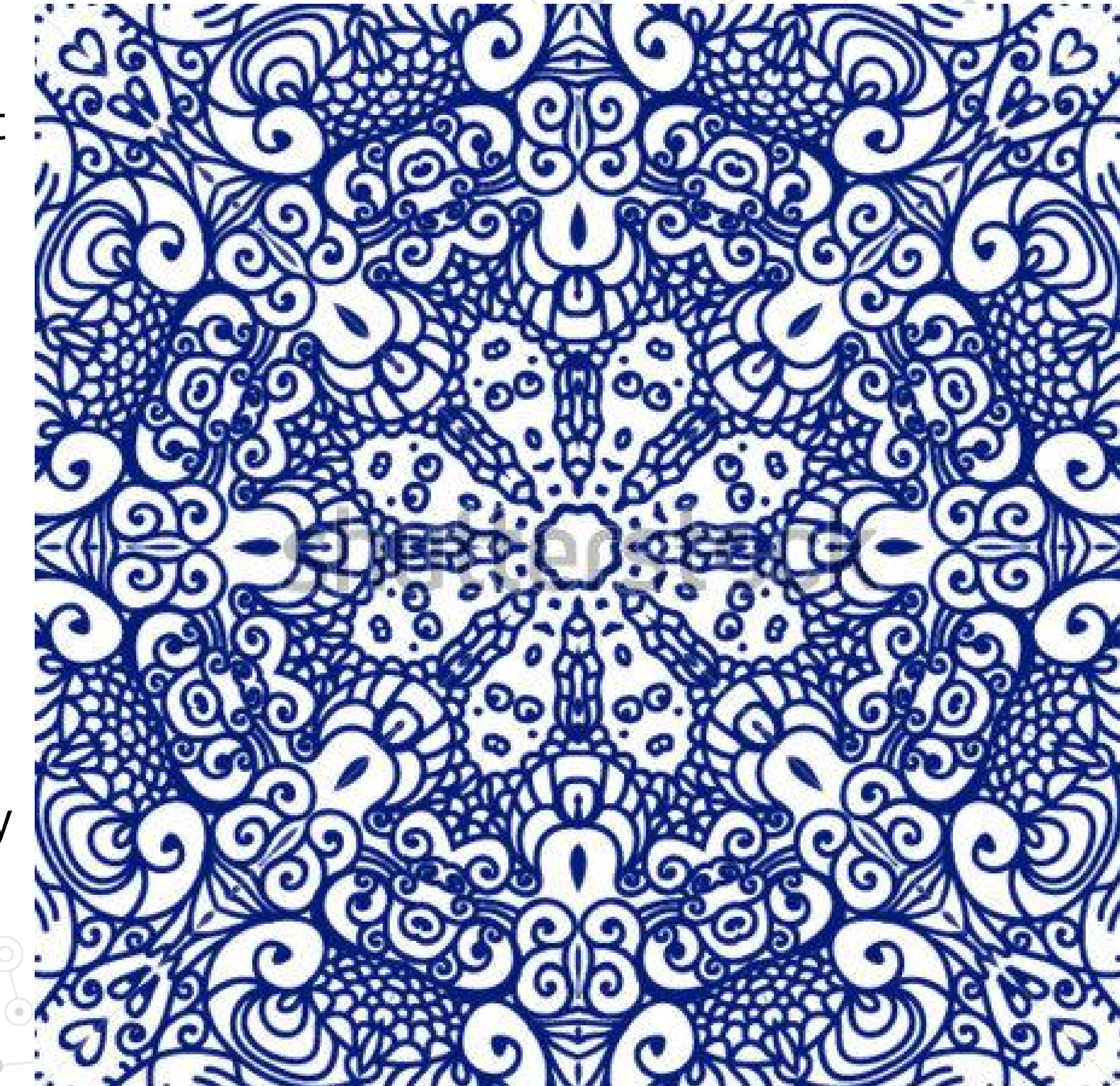
1.

Textures

Map
Could be an image over a surface, but it’s not the only use for it, there are many more uses.

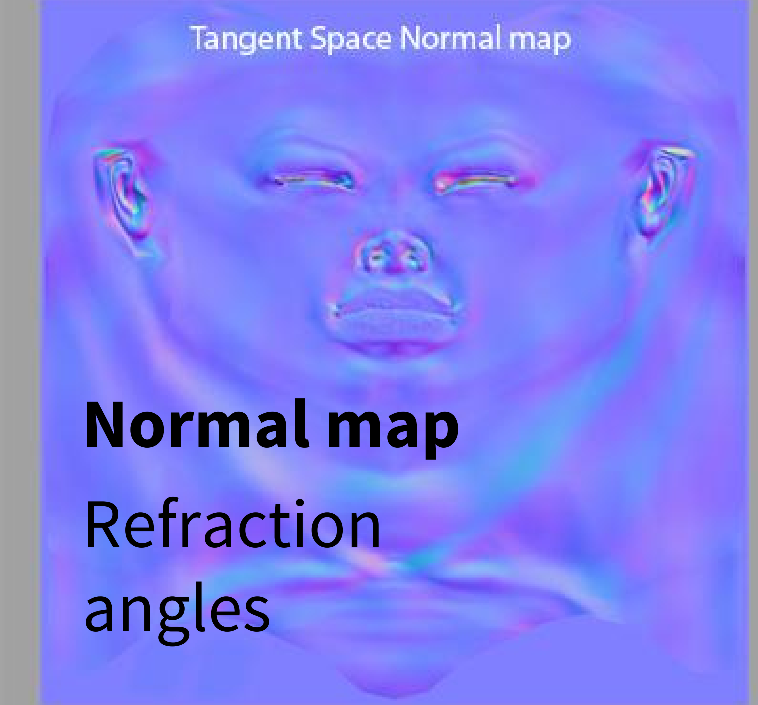
Color map

Texture that is mapped over a surface, usually to give the appearance of having a more complex makeup than it really has.



Maya viewport render, color+normal+specular maps

Color map
Color map


Tangent Space Normal map
Normal map
Refraction angles

Specular color map
Specular map
Mask highlights

Ambient occlusion map

Global illumination occluded places.

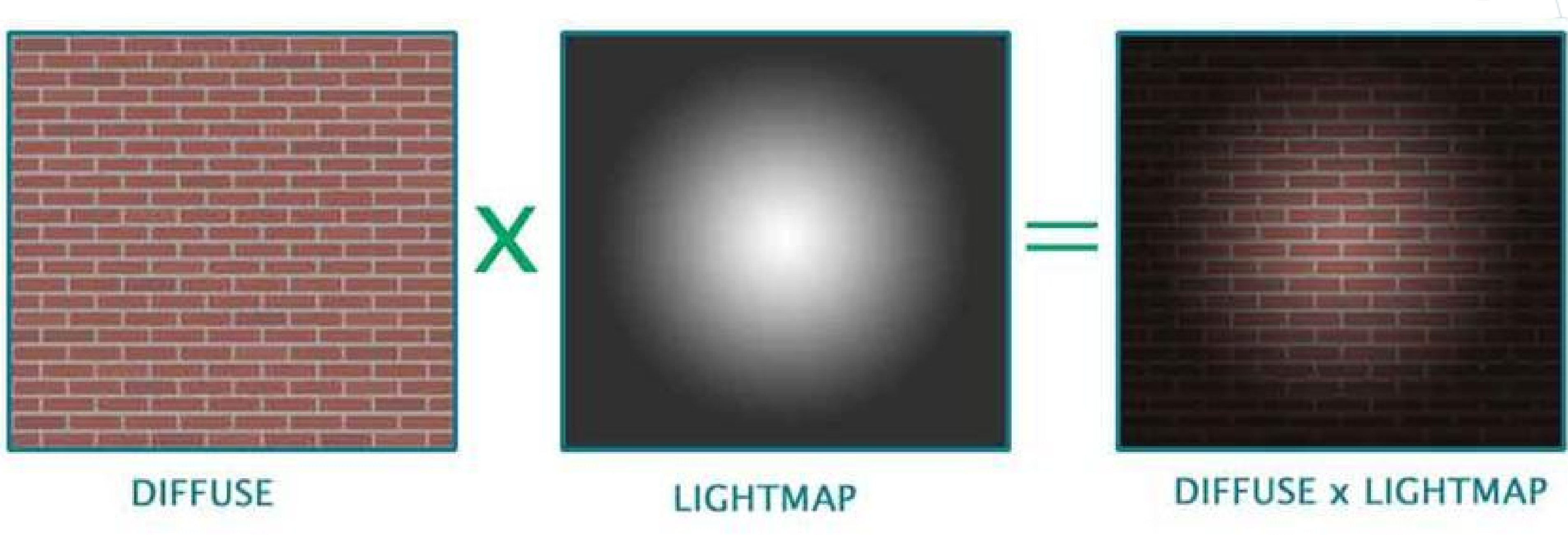
Original model

With ambient occlusion

Extracted ambient occlusion map

Light map

Prerendered lights and shadows.



Shadow map

Used during real-time generation and rendering.

Sample Range: 0,01
Size: 512

Sample Range: 0,01
Size: 2048

Sample Range: 0,01
Size: 4096

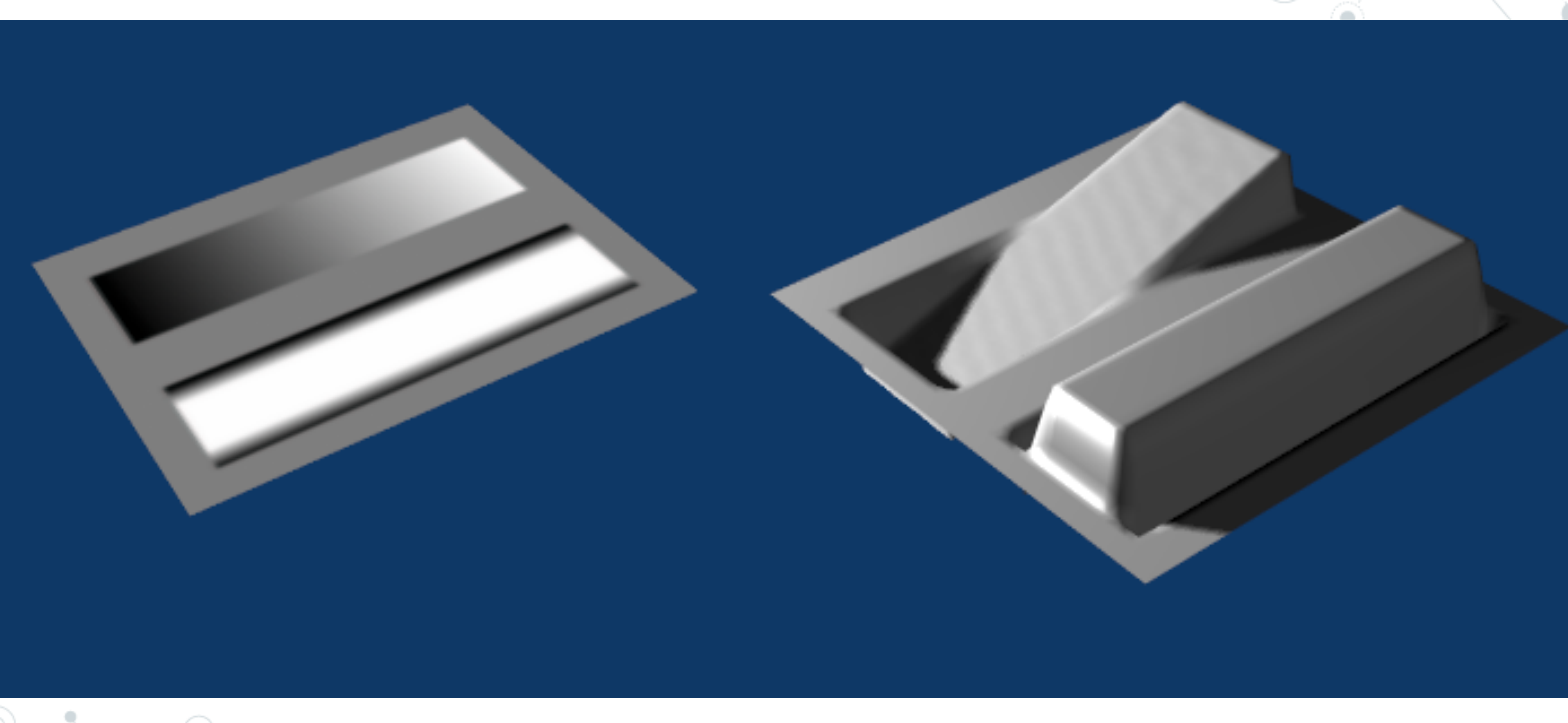
Sample Range: 0,01
Size: 2048

Sample Range: 2,0
Size: 2048

Sample Range: 6,0
Size: 2048

Displacement map

Used for deform the geometry.



Cubemap

Used for reflections and environment lighting.

Alpha map

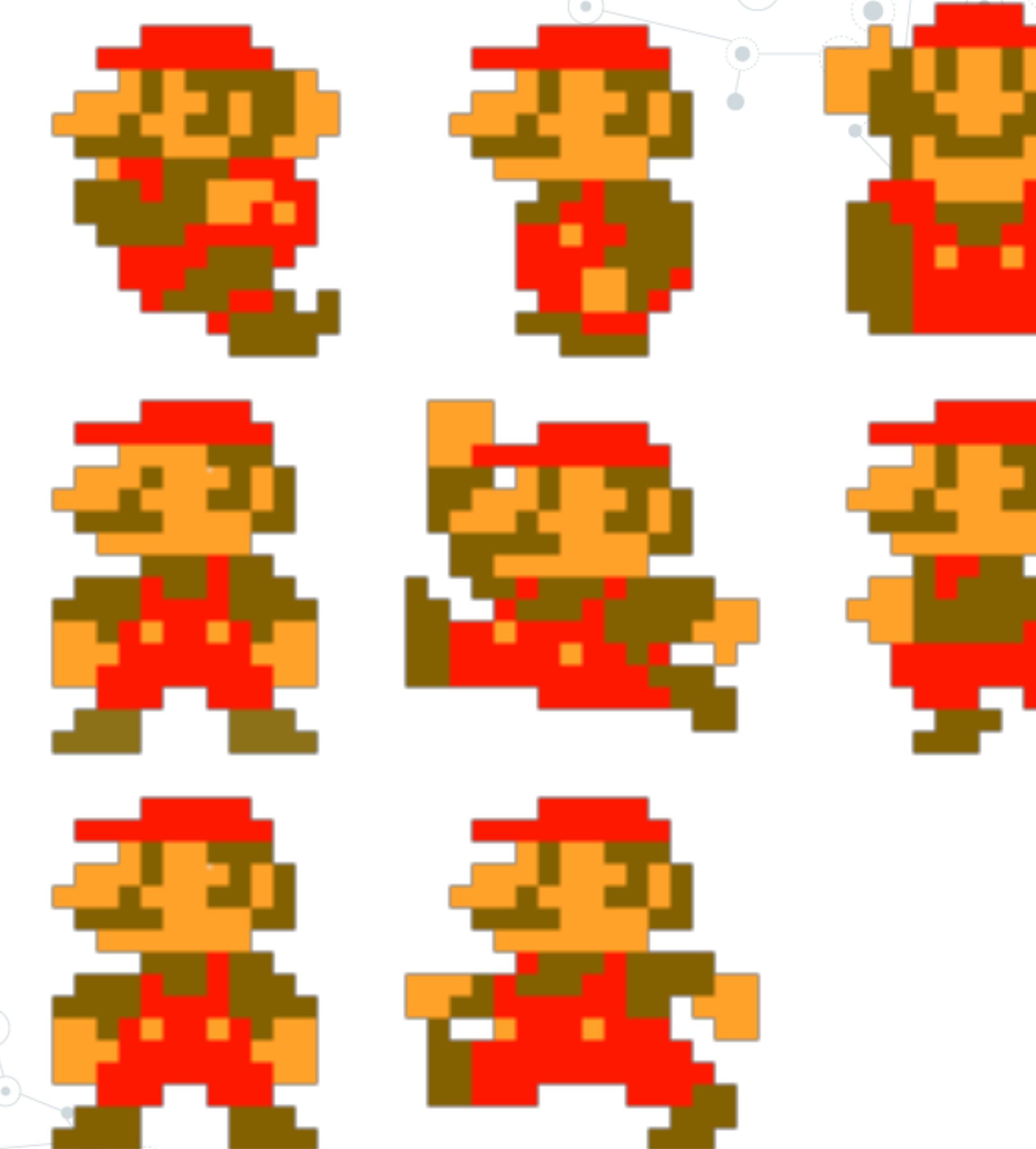
Used for transparency.

Diffuse Map + Alpha Map = Isolated Object

2.
Sprites

Sprite

2D graphical element that appears on the screen.



Billboard

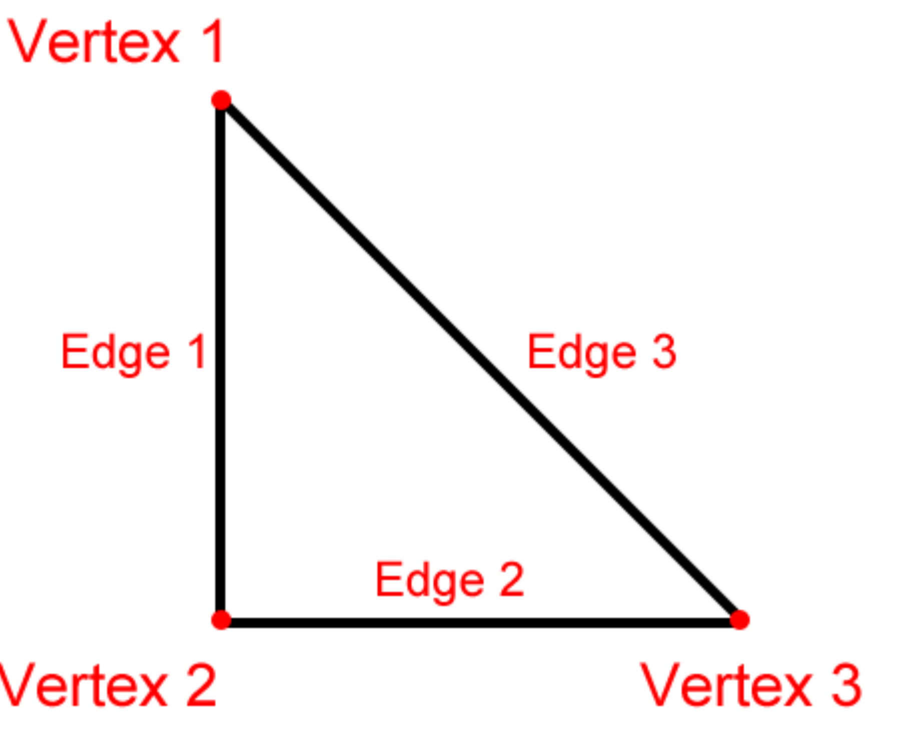
Sprite that is always facing the camera.

3.

3D Graphics Geometry

Vertex

Shapes are defined in game graphics as a collection of points connected together via lines whose insides are shaded in by the graphics hardware. These points are referred to as vertices (the plural term for a vertex), and these lines are referred to as edges.



Triangle

Every 3D model can be described as a group of contiguous polygons.

Each polygon is a shape that can be composed by a group of contiguous triangles.

Mesh

Collection of triangles, describing a part of a 3D model.

Basic Primitive Types

Triangle information on 3D graphics, and other primitive types, are stored as array of vertices.

This array could be interpreted in different ways:

- Point list
- Line list
- Line strip
- Triangle list
- Triangle strip
- Triangle fan

Point lists

Is a collection of vertices that are rendered as isolated points.

Can use them in 3D scenes for star fields, or dotted lines on the surface of a polygon.

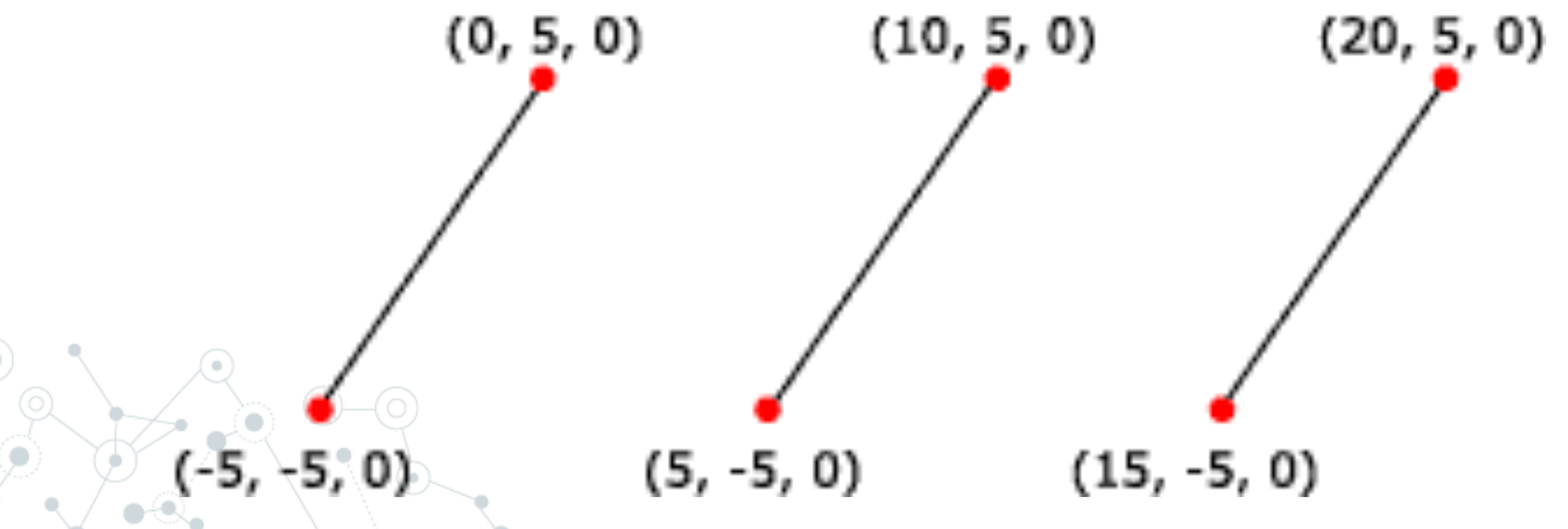
(0, 5, 0) (10, 5, 0) (20, 5, 0)

(-5, -5, 0) (5, -5, 0) (15, -5, 0)

Line lists

Is a list of isolated, straight line segments.

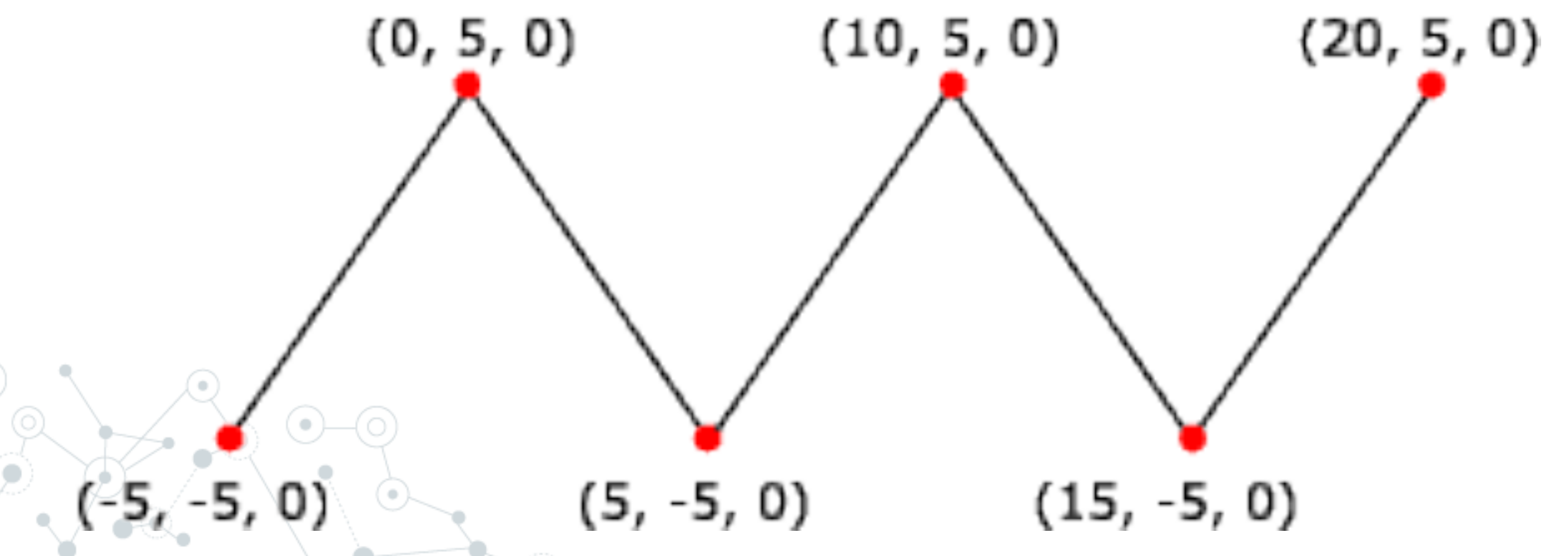
Line lists are useful for such tasks as adding sleet or heavy rain to a 3D scene. Applications create a line list by filling an array of vertices.



Line strips

A line strip is a primitive that is composed of connected line segments.

Can be used for creating polygons that are not closed.

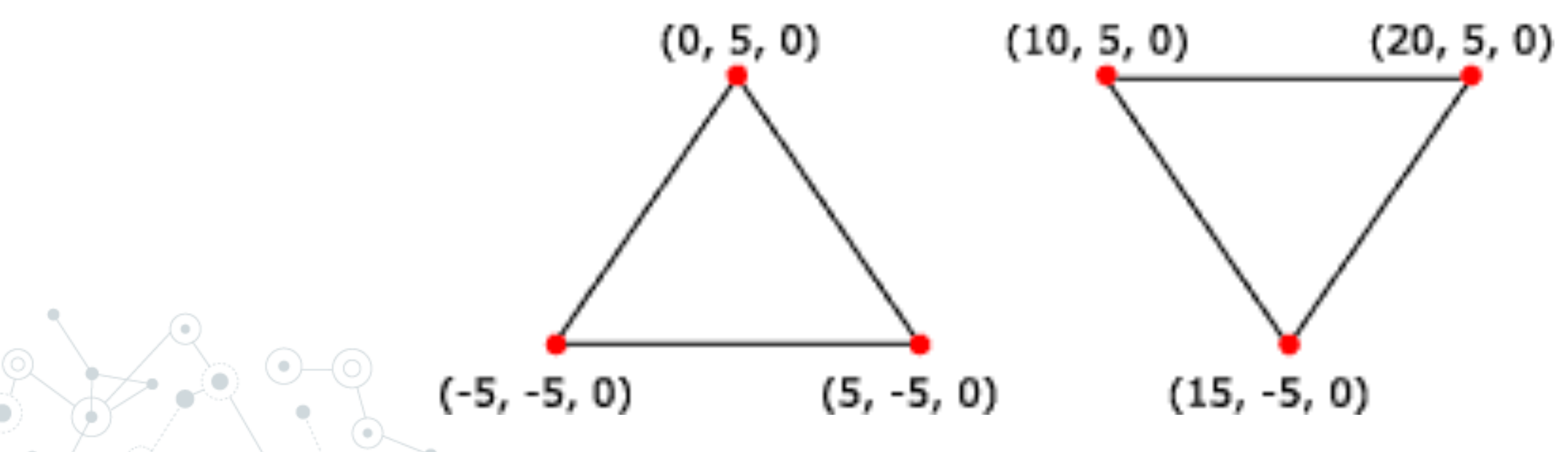


Triangle list

A triangle list is a list of isolated triangles.

They might or might not be near each other.

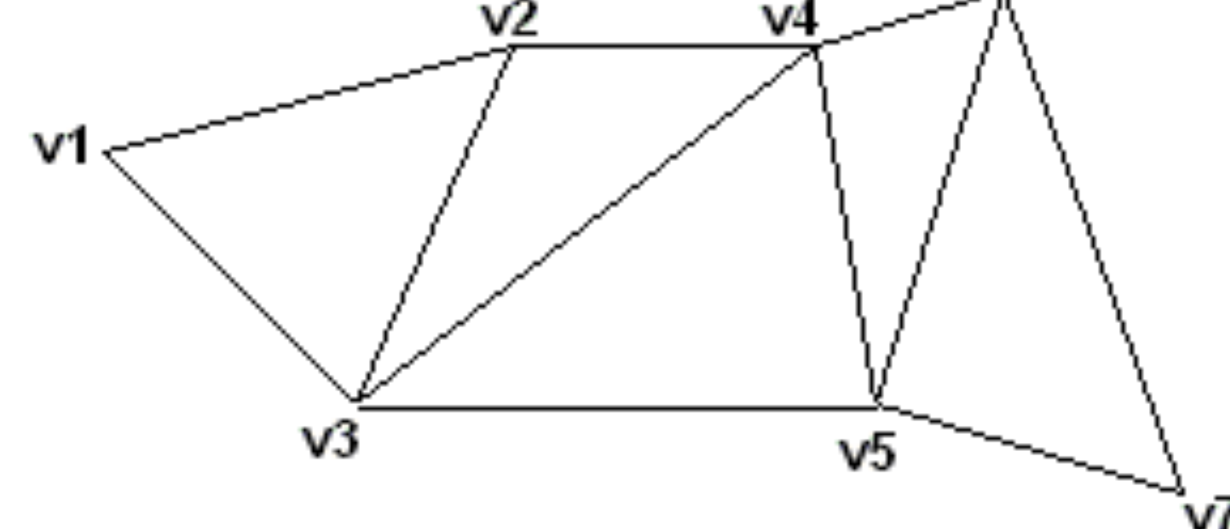
A triangle list must have at least three vertices and the total number of vertices must be divisible by three.



Triangle strip

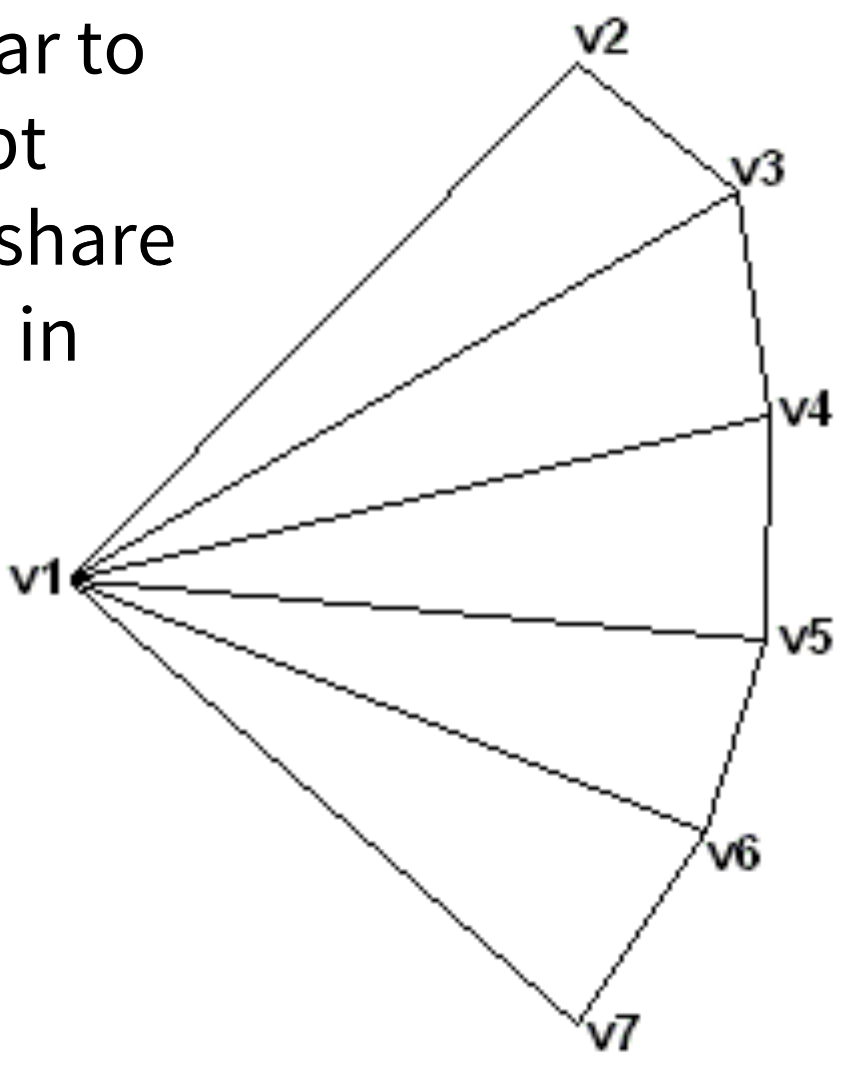
A triangle strip is a series of connected triangles.

The first three vertices define the first triangle and the fourth vertex, along with the previous two vertices define the second triangle and so on.



Triangle fan

A triangle fan is similar to a triangle strip, except that all the triangles share one vertex, as shown in the illustration.



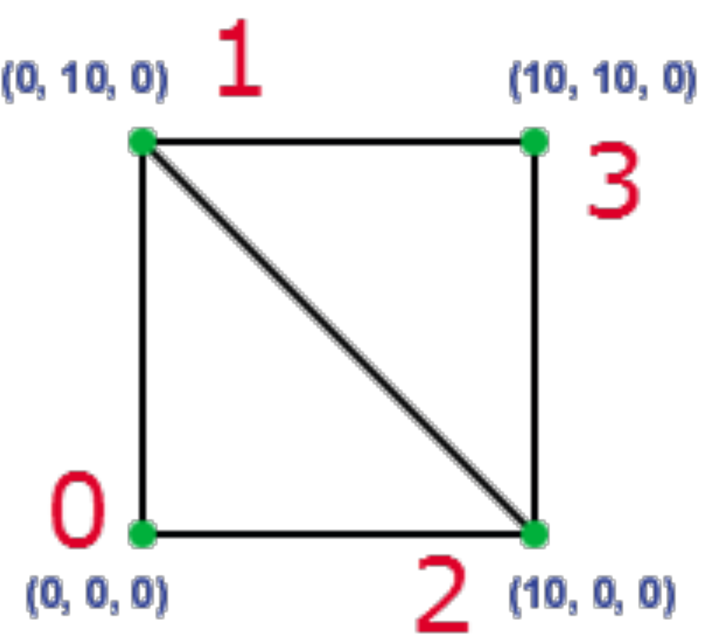
Triangle fan is not more supported

Since DirectX 10 triangle fan is no more an option on the basic primitives, because are not cache friendly.

This is because for drawing a triangle fan, is necessary to continuously jumping back to the first element on the list.

Indexed geometry

Using only unique vertices on the array of vertices and keeping on other array indexes of that list to define which points make up which triangle.

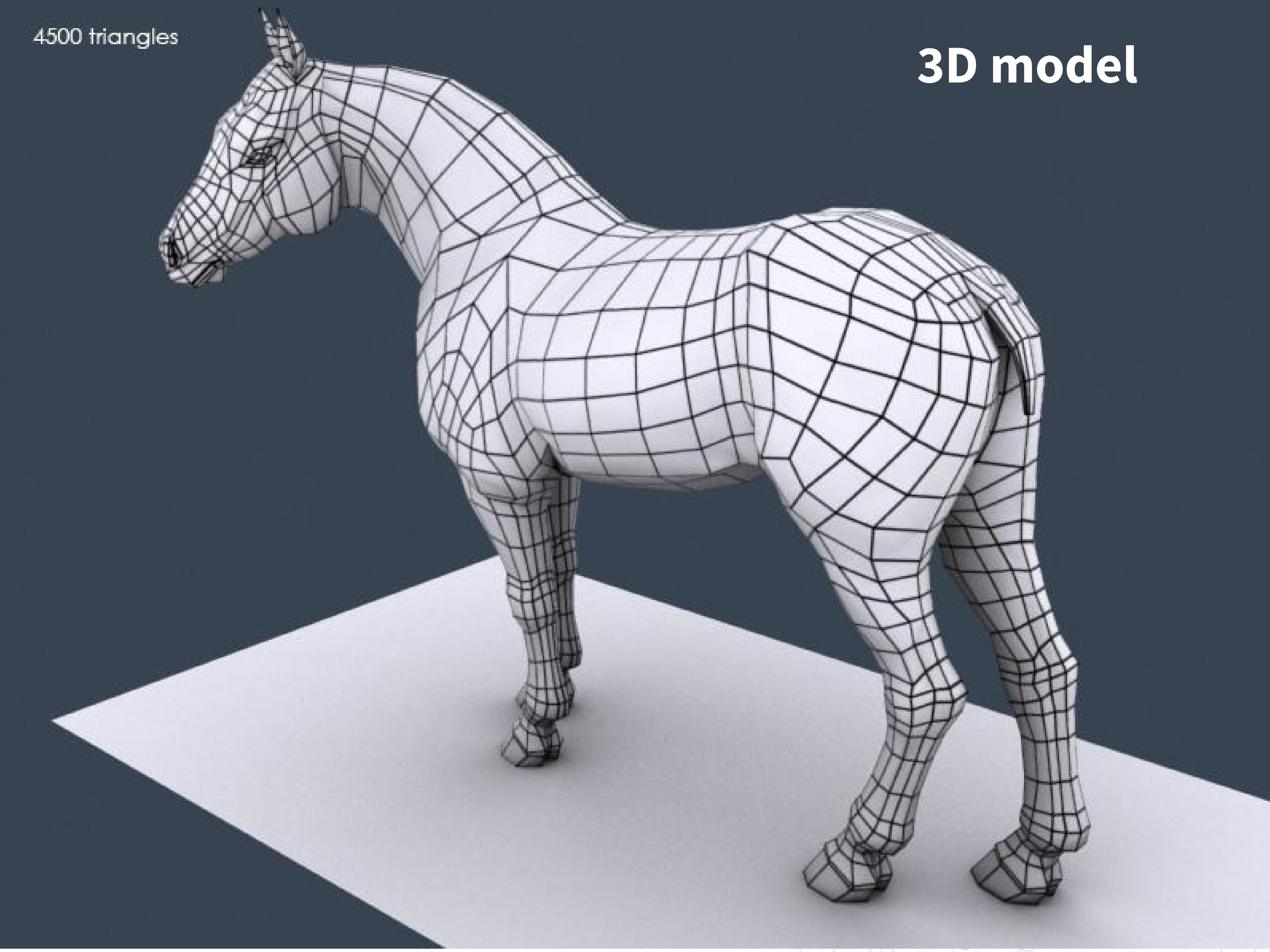


Vertex Buffer

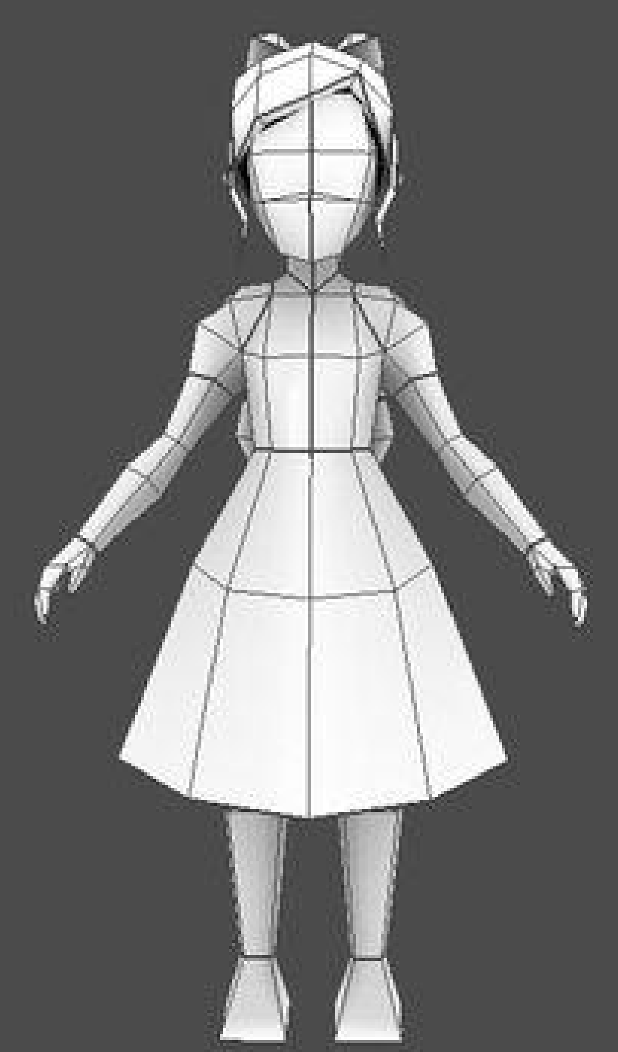
[0] = (0, 0, 0)
[1] = (0, 10, 0)
[2] = (10, 0, 0)
[3] = (10, 10, 0)

Index Buffer

[0] = 0
[1] = 1
[2] = 2
[3] = 3
[4] = 2
[5] = 1

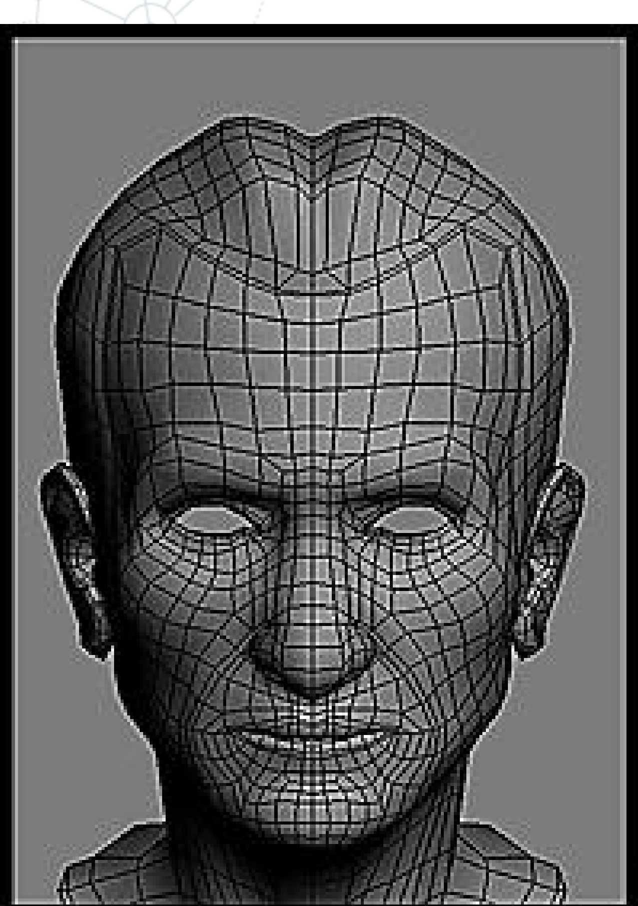

4500 triangles
3D model

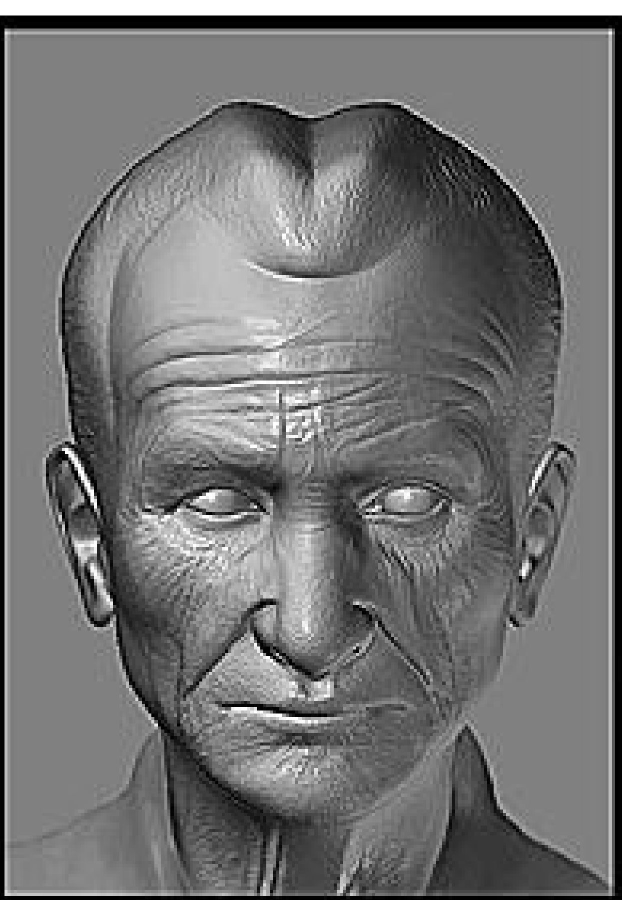
Lowpoly



642 Tris
512 Diffuse
Little Sister Concept © 2K Games

3D model advanced techniques


Base Mesh created in 3ds max


Detailing Mesh in Zbrush

Hair corner adjust and Background in Photoshop, Rendering and texturing in Zbrush

Vertex properties

Each vertex has a host of information the shaders will need to produce an effect.

Most common properties are:

- Position
- Color
- Normal
- Texture coordinates
- Bone weights and indices

Pixel data

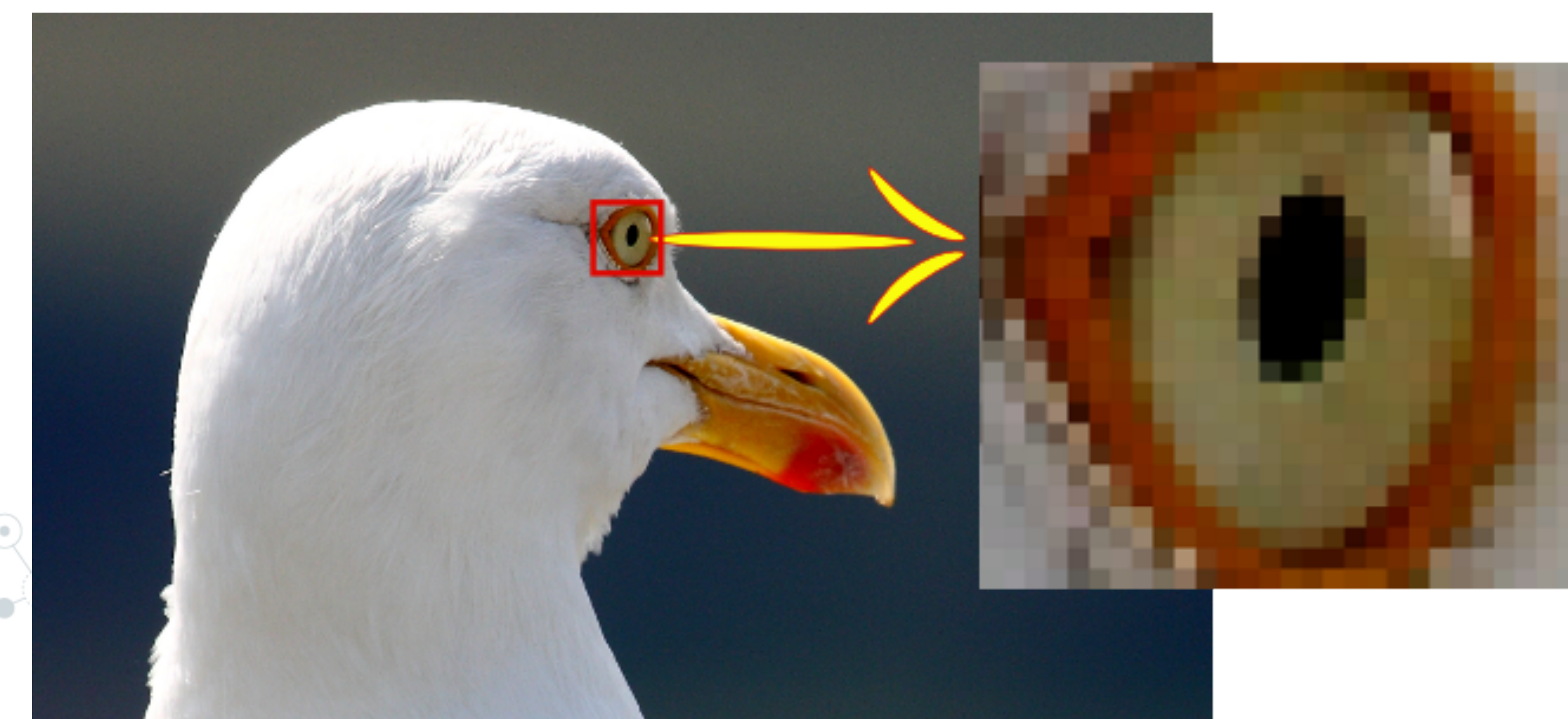
Per-pixel data is calculated using interpolation.

The pixel shader receives interpolated data from the vertex shader (or geometry shader if is present).

This includes positions, colors, texture coordinates and all other attributes provided by the previous shader stage.

Pixel data

If is necessary to specify per-pixel data, it is often in the form of texture images, such as a normal map texture.



4.

Vertex Buffers

Vertex Buffers

A buffer is memory of a specific size. For example a buffer of 40 bytes is necessary for storing 10 integers (if each integer = 4 bytes)

Vertex buffer is a Direct3D buffer of type **ID3D11Buffer** that is used to store all the vertex data for a mesh.

Vertex Buffers

These buffers resides in an optimal location of memory, such the GPU, which is chosen by the device driver.

Most advanced 3D commercial video games use techniques to determine what geometry is visible beforehand and only submit those that are either visible or potentially visible to the graphics hardware.

Vertex Buffer Creation

```
struct VertexPos
{
   XMFLOAT3 pos;
};

VertexPos vertices[] =
{
   XMFLOAT3( 0.5f, 0.5f, 0.5f ),
   XMFLOAT3( 0.5f, -0.5f, 0.5f ),
   XMFLOAT3( -0.5f, -0.5f, 0.5f )
};
```

Vertex Buffer Creation

```
D3D11_BUFFER_DESC vertexDesc;
ZeroMemory( &vertexDesc, sizeof( vertexDesc ) );
vertexDesc.Usage = D3D11_USAGE_DEFAULT;
vertexDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vertexDesc.ByteWidth = sizeof( VertexPos ) * 3;

D3D11_SUBRESOURCE_DATA resourceData;
ZeroMemory( &resourceData, sizeof(resourceData) );
resourceData.pSysMem = vertices;

ID3D11Buffer* vertexBuffer;
HRESULT result = d3dDevice_->CreateBuffer(
   &vertexDesc, &resourceData, &vertexBuffer );
```

Vertex Buffer Creation

Buffer descriptor. Buffer details, because could be created another type of buffer than a vertex buffer.

Sub resources. Used for pass the vertex data to the buffer.

Create buffer. Function for finally create the buffer.

5.

Input Layout

Input Layout

Vertex buffer is a chunk of data, and we need to specify the attributes, the ordering and size.

Input Layout is used for this task, with:

`D3D11_INPUT_ELEMENT_DESC`

Input Layout

```
typedef struct D3D11_INPUT_ELEMENT_DESC {
   LPCSTR SemanticName;
   UINT SemanticIndex;
   DXGI_FORMAT Format;
   UINT InputSlot;
   UINT AlignedByteOffset;
   D3D11_INPUT_CLASSIFICATION InputSlotClass;
   UINT InstanceDataStepRate;
} D3D11_INPUT_ELEMENT_DESC;
```

Input Layout

SemanticName. Describes the purpose of the element: POSITION, COLOR,...

SemanticIndex. Vertex can use multiple elements with same semantic (texture coordinates, for example)

Format. For example DXGI_FORMAT_R32G32B32_FLOAT could be used for position with 4 bytes (32 bits) floating-poing axes

Input Layout

Input Slot. Which vertex buffer the element is found in.

Aligned byte offset. Starting byte offset into the vertex buffer where it can find this element.

Input slot class. Describe whether the element is to be used for each vertex of for each instance (per object), used for draw multiple objects with a single draw call.

Input Layout

<u>Instance data step rate</u>. Which is used to tell how many instances to draw in the scene.

Input layouts are created using:

```
HRESULT CreateInputLayout(
   const D3D11_INPUT_ELEMENT_DESC*
                 pInputElementDescs,
   UINT NumElements,
   const void*
      pShaderBytecodeWithInputSignature,
   SIZE_T BytecodeLength,
   ID3D11InputLayout** ppInputLayout
);
```

Input Layout creation

pInputElementDescs y NumElements. Are the array of elements in the vertex layout and number of elements in that array.

pShaderBytecodeWithInputSignature. Is the compiled vertex shader code.

BytecodeLength. Is the size of the shader's bytecode. The vertex shader's input signature must match our input layout.

ppInputLayout. Is the pointer of the object that will be created with this function call.

6.

Basic Shaders

Basic shaders

Shaders that are not optional are the pixel and vertex:

```
ID3D11VertexShader* solidColorVS;
ID3D11PixelShader* solidColorPS;
```

Vertex shader is needed before creating the input layout, since vertex shader’s signature must match the input layout.

Shader compilation

Compiled code executed on GPU, written on HLSL and could be load & compile with:

```
HRESULT D3DX11CompileFromFile(
    LPCTSTR pSrcFile,
    const D3D10_SHADER_MACRO* pDefines,
    LPD3D10INCLUDE pInclude,
    LPCSTR pFunctionName,
    LPCSTR pProfile,
    UINT Flags1,
    UINT Flags2,
    ID3DX11ThreadPump* pPump,
    ID3D10Blob** ppShader,
    ID3D10Blob** ppErrorMsgs,
    HRESULT* pHResult
);
```

HLSL

HLSL is the High Level Shading Language for DirectX, used to create C like programmable shaders.

Was created on DirectX 9 to set up the programmable 3D pipeline.

Pipeline can be programmed using a combination of assembly instructions, HLSL instructions and fixed-function statements.

Vertex & Pixel shader creation

With compiled shader, a vertex shader can be created with:

```
HRESULT CreateVertexShader(
   const void* pShaderBytecode,
   SIZE_T BytecodeLength,
   ID3D11ClassLinkage* pClassLinkage,
   ID3D11VertexShader** ppVertexShader
);
```

CreatePixelShader has the same signature.

Basic Vertex shader

A basic example of a vertex shader:

```
float4 VS( float4 pos : POSITION ) : SV_POSITION
{
   return pos;
}
```

SV stands for System-value, which can’t be modify by CPU. Are input / output from a shader stage or are generated entirely by the GPU.

Basic Pixel shader

A basic example of a pixel shader:

```
float4 PS( float4 pos : SV_POSITION ) : SV_TARGET
{
   return float4( 0.0f, 1.0f, 0.0f, 1.0f );
}
```

Note that the input is a position on 3D space (SV_POSITION) and the output is a color value (SV_TARGET)

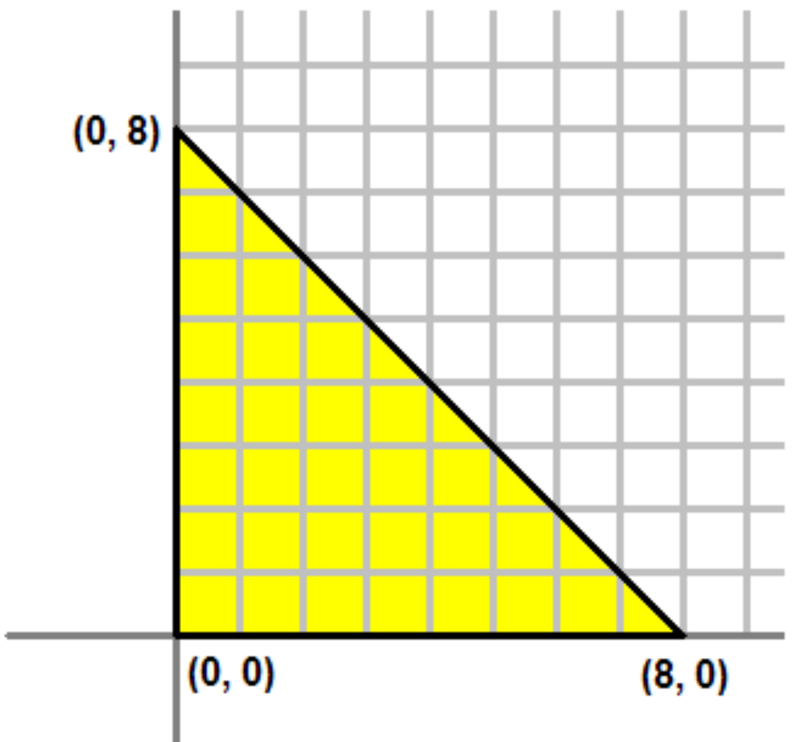
Pixel limitations

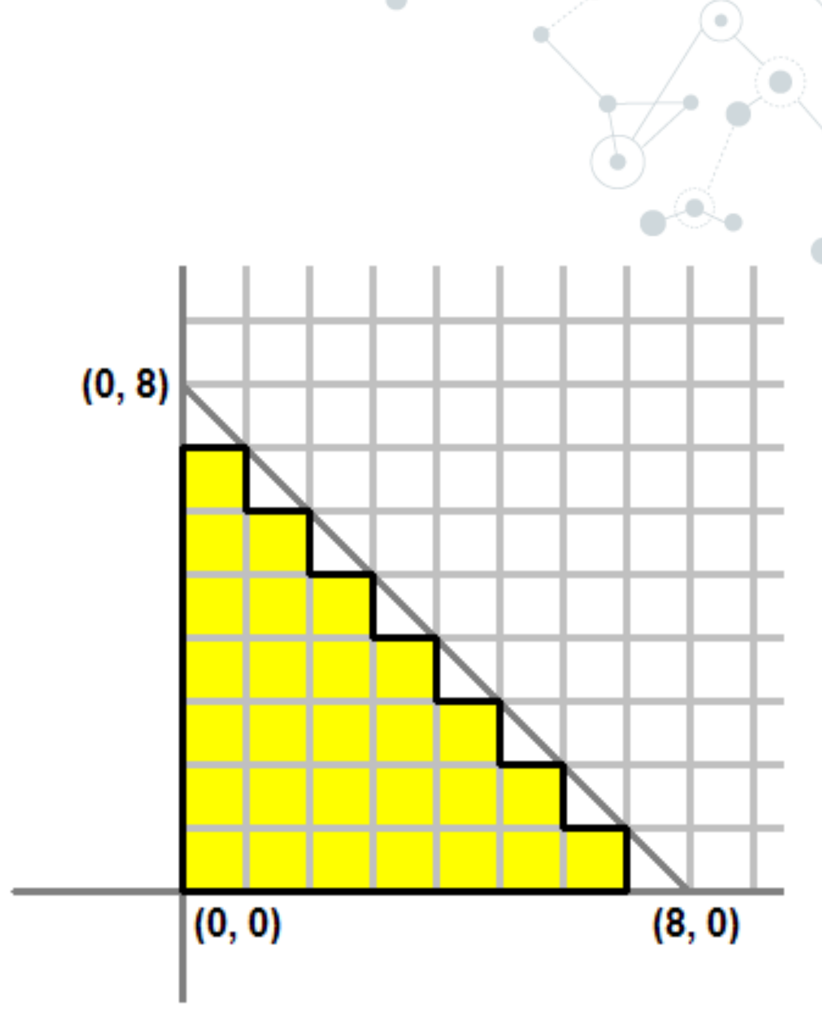
Modern computer monitors are commonly raster display, which means the screen is actually a two-dimensional grid of small dots called pixels.

Each pixel contains a color independent of other pixels.

When we render a triangle on the screen, we don't really render a triangle as one entity. Rather, we light up the group of pixels that are covered by the triangle's area.

Pixel limitations





7.

Rendering

Input Assembler

For rendering in Direct3D, we need to set up the input assembly, bind our shaders and other assets (such as textures), and draw each mesh.

To set up Input Assembly is used:

```
IASetInputLayout

IASetVertexBuffers

IASetPrimitiveTopology
```

IASetInputLayout

Used to bind the vertex layout that we created when we called the device's `CreateInputLayout`.

This is done each time we are about to render geometry that uses a specific input layout.

IASetVertexBuffers

Used to set one or more vertex buffers:

```
void IASetVertexBuffers(
   UINT StartSlot,
   UINT NumBuffers,
   ID3D11Buffer* const* ppVertexBuffers,
   const UINT* pStrides,
   const UINT* pOffsets
);
```

IASetPrimitiveTopology

Used to tell Direct3D what type of geometry we are rendering, for example:

- `D3D11_PRIMITIVE_TOPOLOGY_POINTLIST`
- `D3D11_PRIMITIVE_TOPOLOGY_LINELIST`
- `D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP`
- `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`
- `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP`

Setting Shaders

After setting the input assembler we can set shaders, this can be done with the functions:

- `VSSetShader`
- `PSSetShader`

Draw

Once we’ve set and bound all of the necessary data for our geometry, we need to call the Draw function:

```
void Draw(
  UINT VertexCount,
  UINT StartVertexLocation
);
```

Present

Last step is calling swap chain’s Present function, that allows us to present the renderer image to the screen:

```
HRESULT Present(
  UINT SyncInterval,
  UINT Flags
);
```