Rendering Pipeline

Ju Heqi （Autodesk Inc. ）

Agenda

- Foundation of rendering
- Rendering pipeline
- Demo

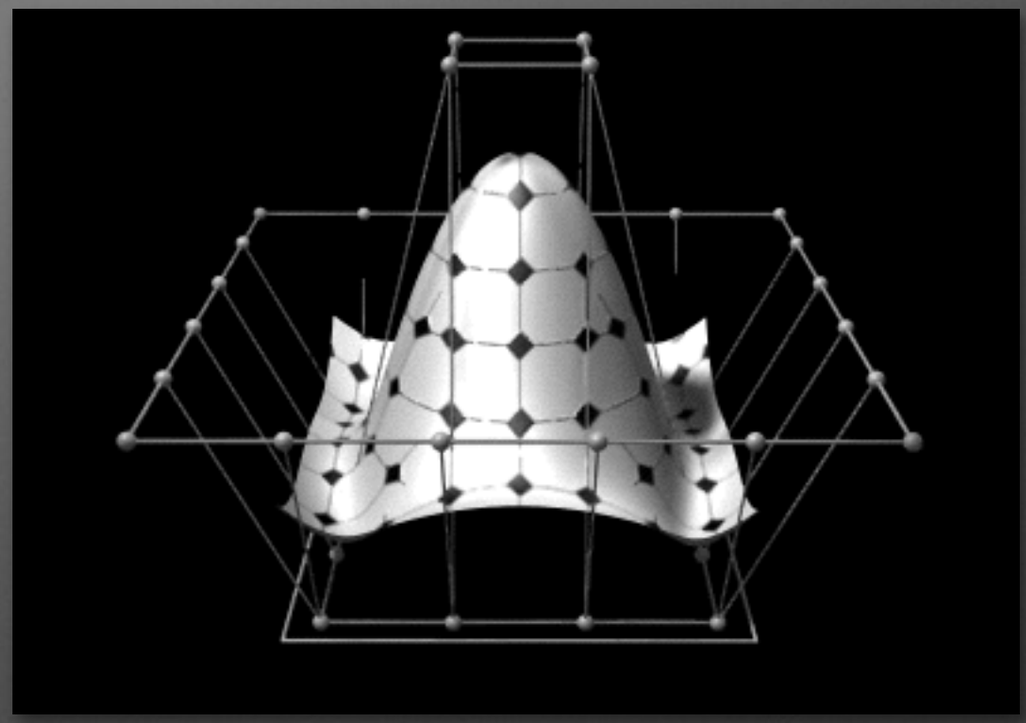
Rendering Foundation

Raw data for this image

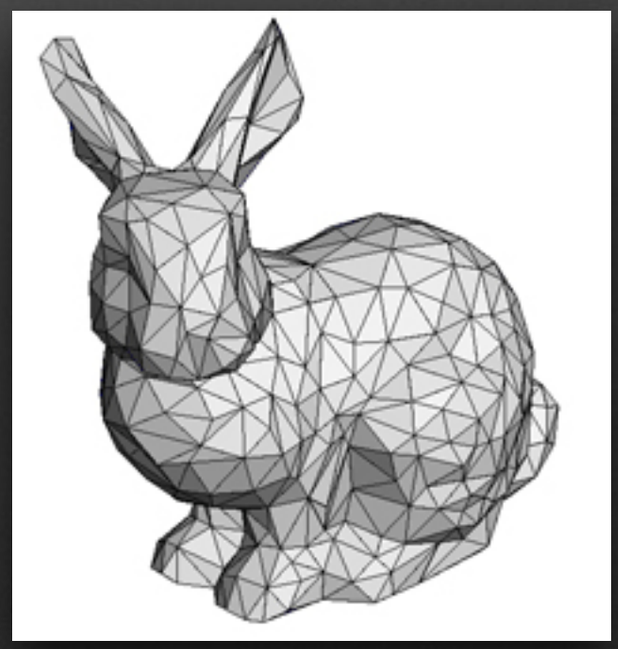
- A virtual scene
- Light source
- Camera
- Visual properties of surface

Describing a Scene

- patches



- triangle meshes



Triangle Meshes: the Pros

- Triangle is the simplest type of polygon
- A triangle is always planar and convex
- Triangle remain triangle under most kind of transformation
- Virtually all commercial GPU hardware is designed around triangle rasterization

Triangle Meshes: The Cons

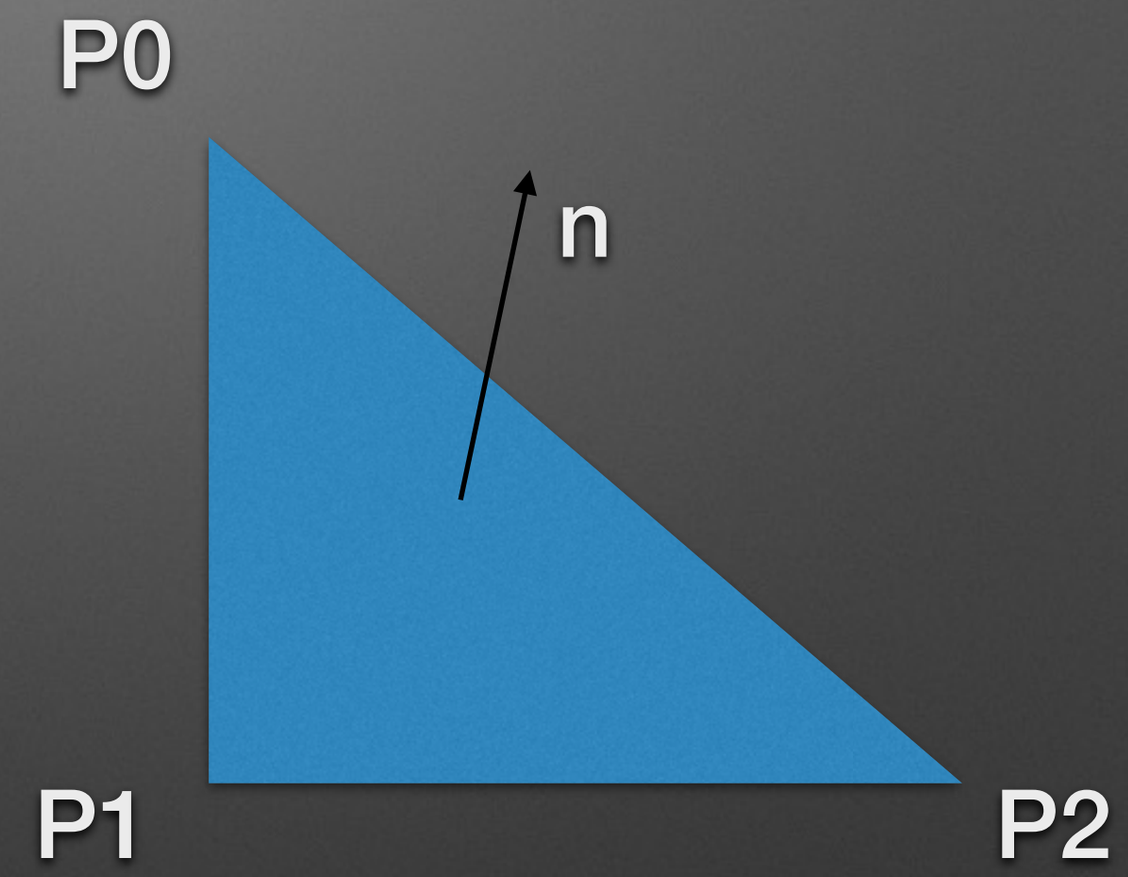
- *tessellation*: a process of dividing a surface up into a collection of discrete polygons

- object's *silhouette edges* looks blocky, when tessellation is low

- performance vs. quality

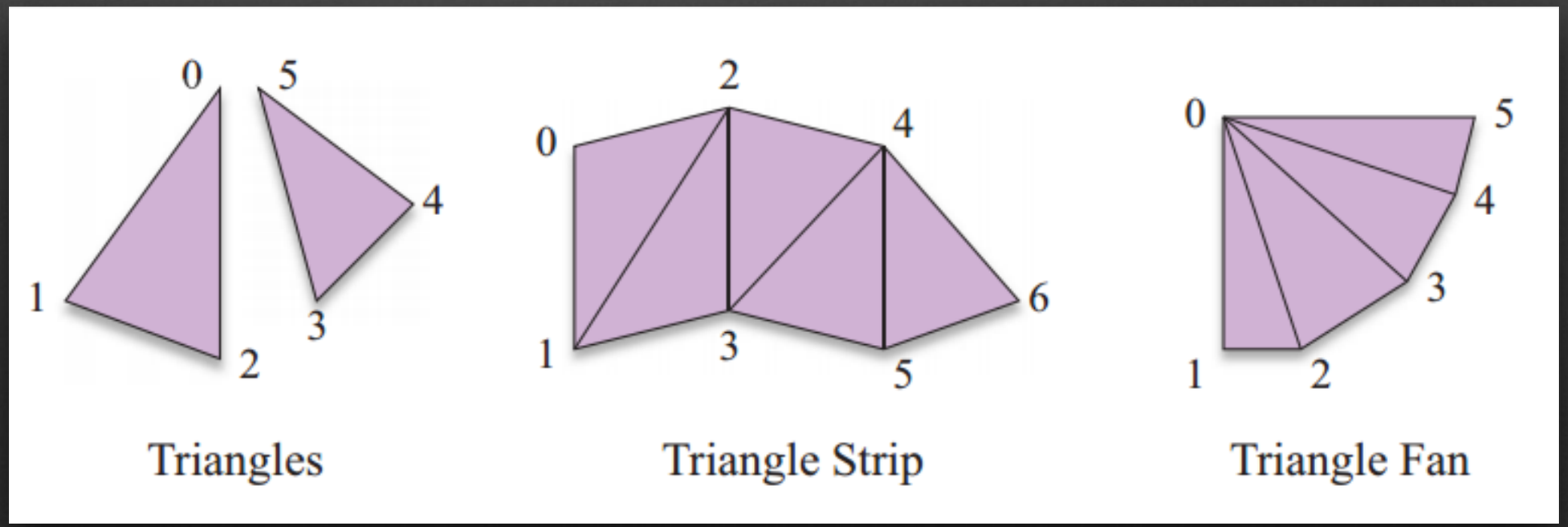
Face normal and Winding Order

- Face Normal
- Winding Order



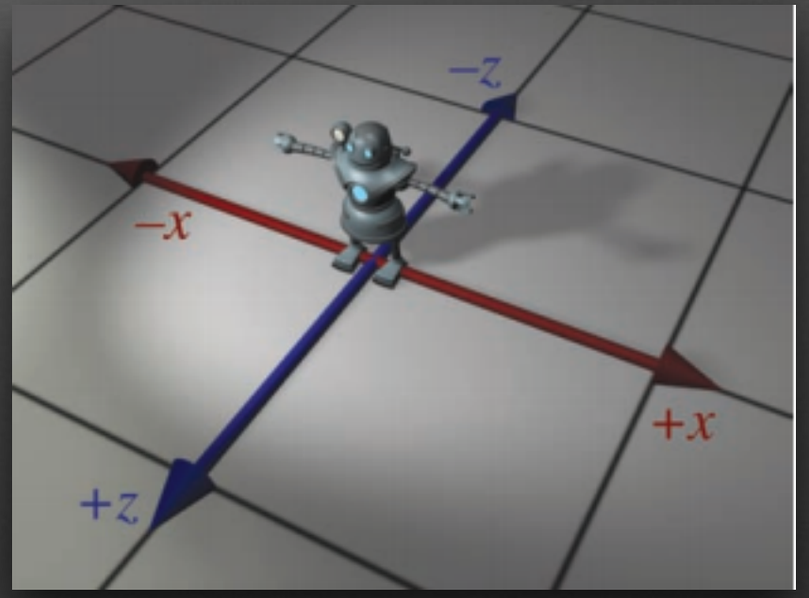
Triangle Mesh Topology

- Triangle Lists
- Indexed Triangle List
- Strips and Fans



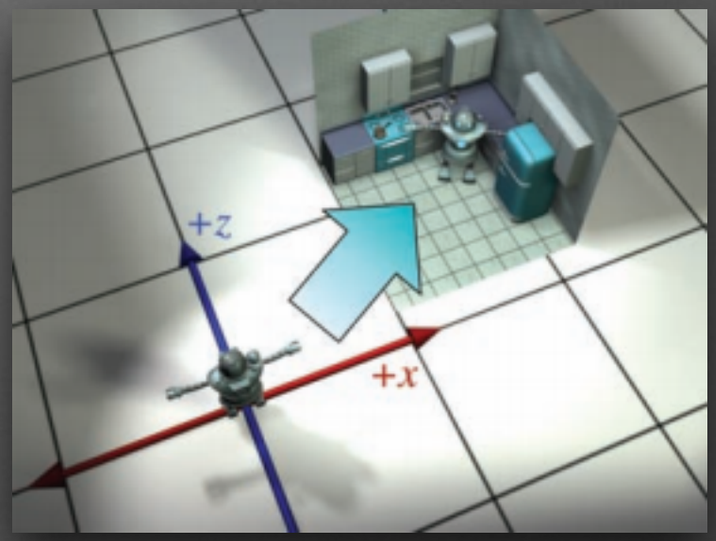
Model Space

- The position vectors of a triangle mesh's vertices are specified relative to a convenient coordinate system called *model space*



World Space

- Many individual meshes are composed into a complete scene by positioning and orienting them within a common coordinate system, known as *world space*



- Model-to-world matrix

$$M_{M\to w} = \begin{bmatrix} (RS)_{M\to W} & 0 \\ t_M & 1 \end{bmatrix}$$

$$M_{M\to w} = \begin{bmatrix} \boldsymbol{i}_M & 0 \\ \boldsymbol{j}_M & 0 \\ \boldsymbol{k}_M & 0 \\ \boldsymbol{t}_M & 1 \end{bmatrix}$$

Describing the Visual Properties of a Surface

- geometric information
  - surface normal
  - tangent & bitangent
- description of how light should interact with the surface
  - diffuse color, etc

Vertex Attributes

- a convenient place to store surface properties
- attributes normally used:
    - position
    - normal
    - tangent / bitangent
    - Diffuse / Specular color
    - Texture coordinates
    - skinning weight

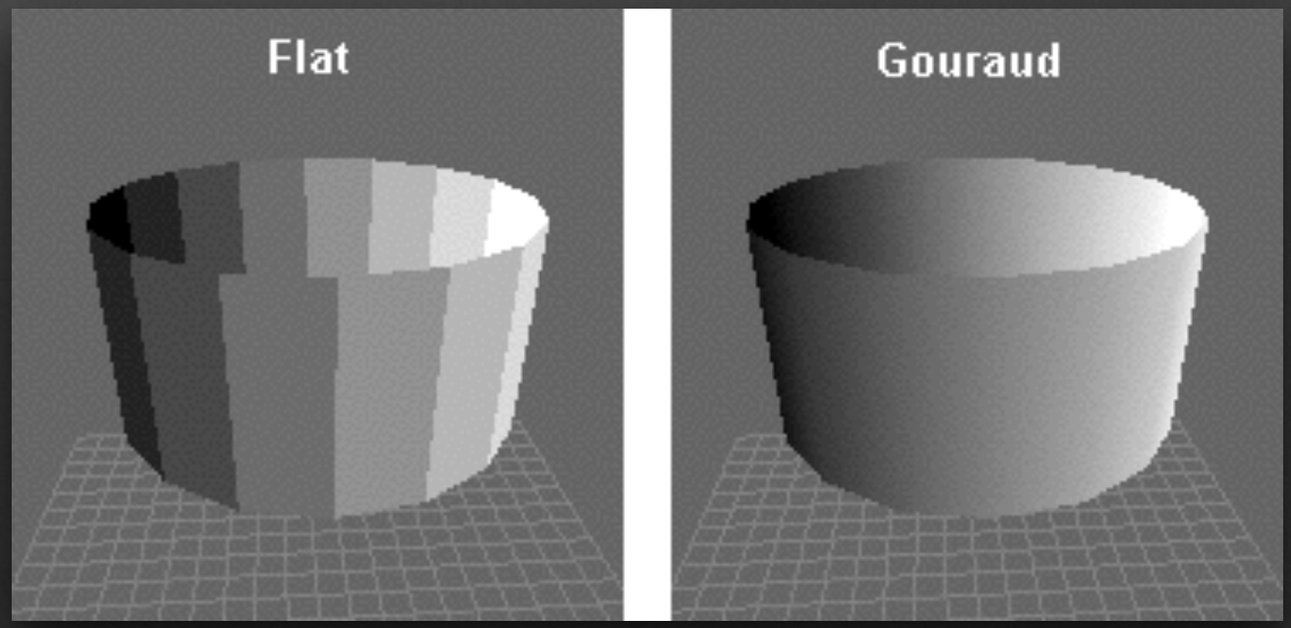
Vertex Format

- *Vertex attributes* are typically stored within a data structure , call *vertex format*

```
struct Vertex {
    Vector3f position;
    Vector3f normal;
    Vector4f color;
    Vector2f texcoords;
};
```

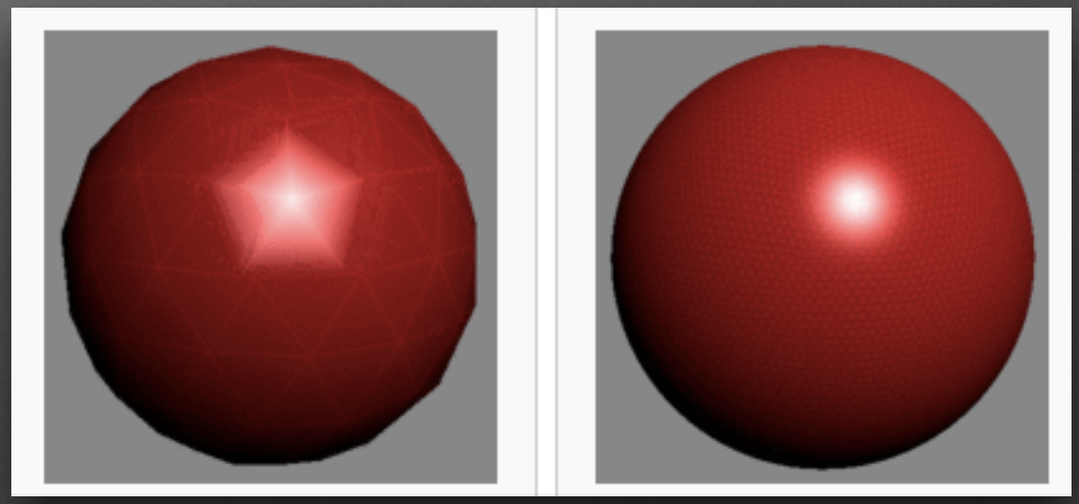
Vertex Attribute Interpolation

- the attributes of vertex is too coarse
- we need to know the values of the attributes on a per-pixel basis not a per-vertex basis
- *Gouraud Shading*



Textures

- Gouraud Shading isn't always good, especially when the tessellation is low



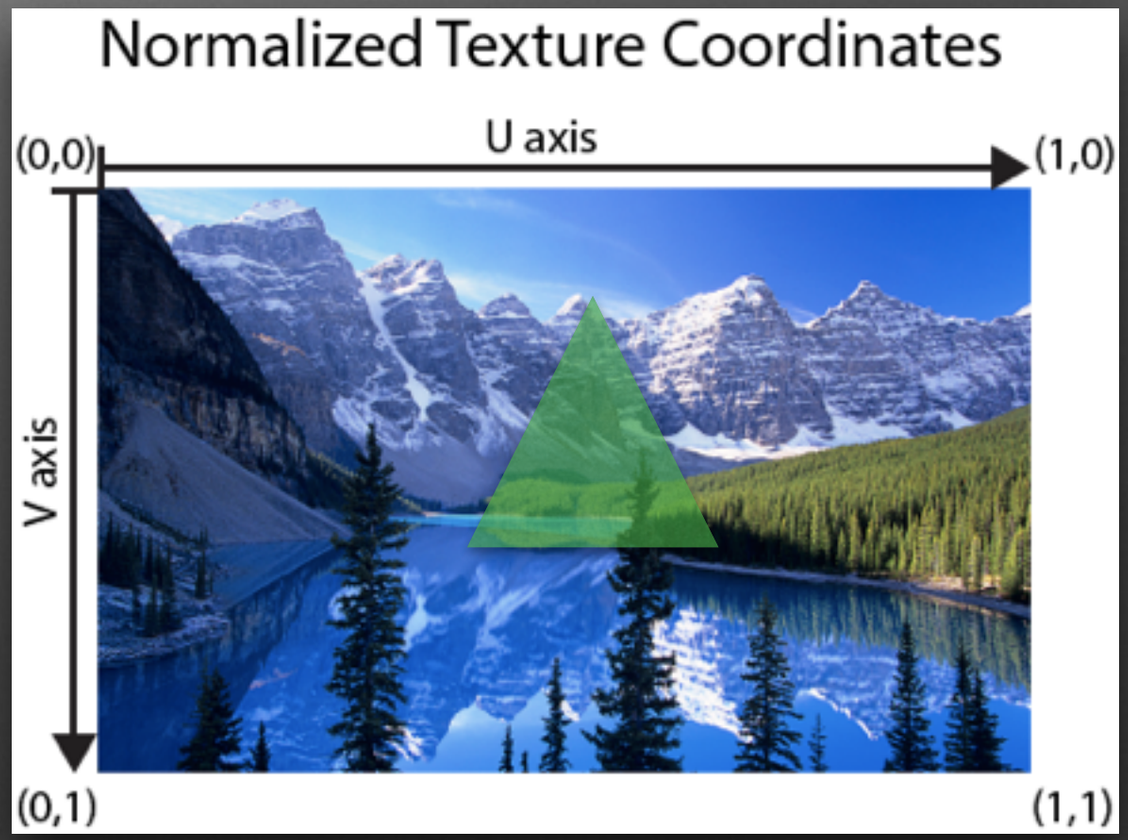
- To overcome the limitation of per-vertex surface attributes, *texture map* is used, so we can control surface color at per-pixel level

Texture Coordinates

- Texture coordinates are usually expressed in *normalized texture space.*

- Traditionally, texture coordinates are call *UV coordinates*

- Texture mapping

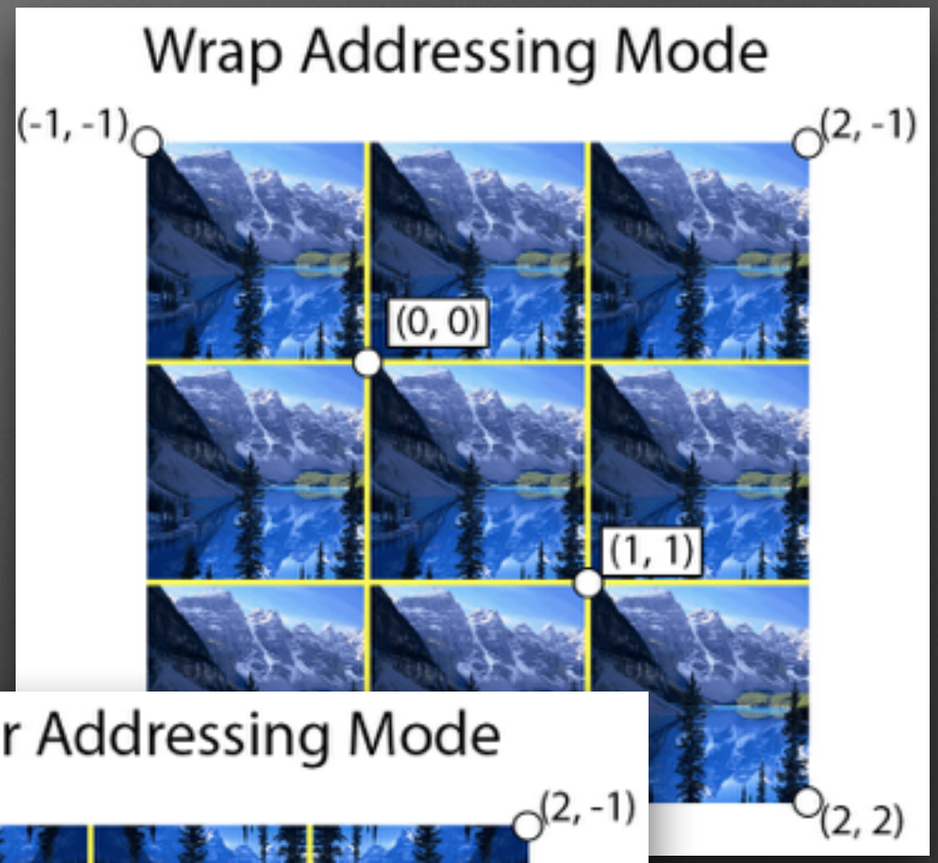


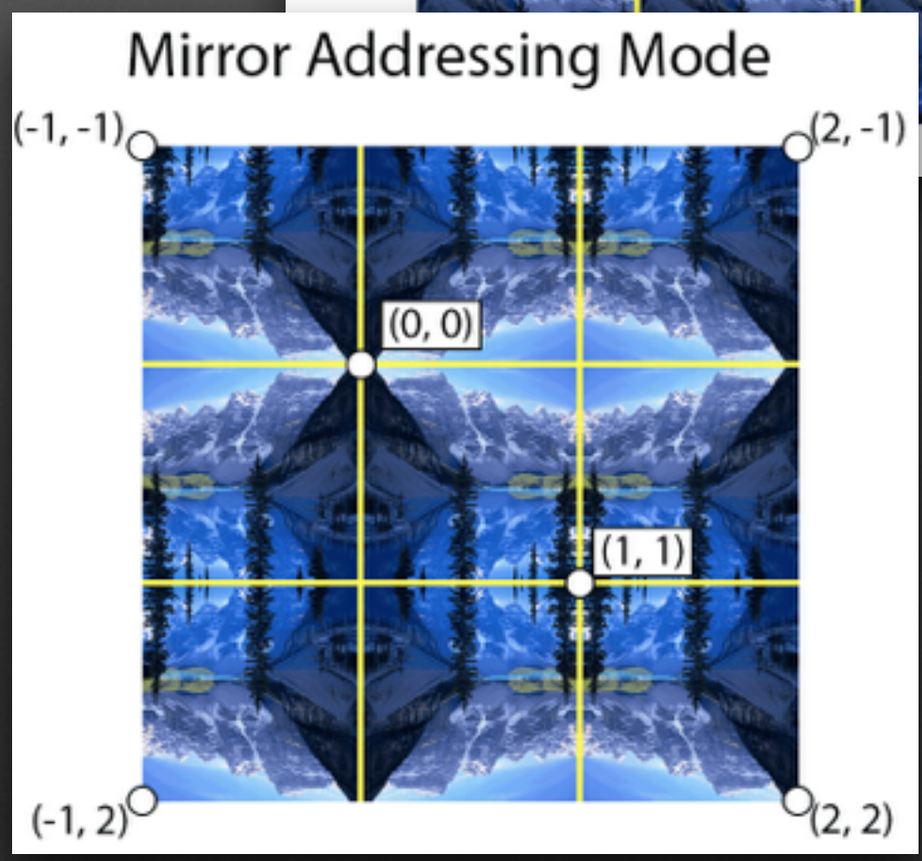
Texture Addressing Mode

- Texture coordinates are permitted to extend beyond the [0, 1] range.
- graphic hardware can handle them

Texture Addressing Mode

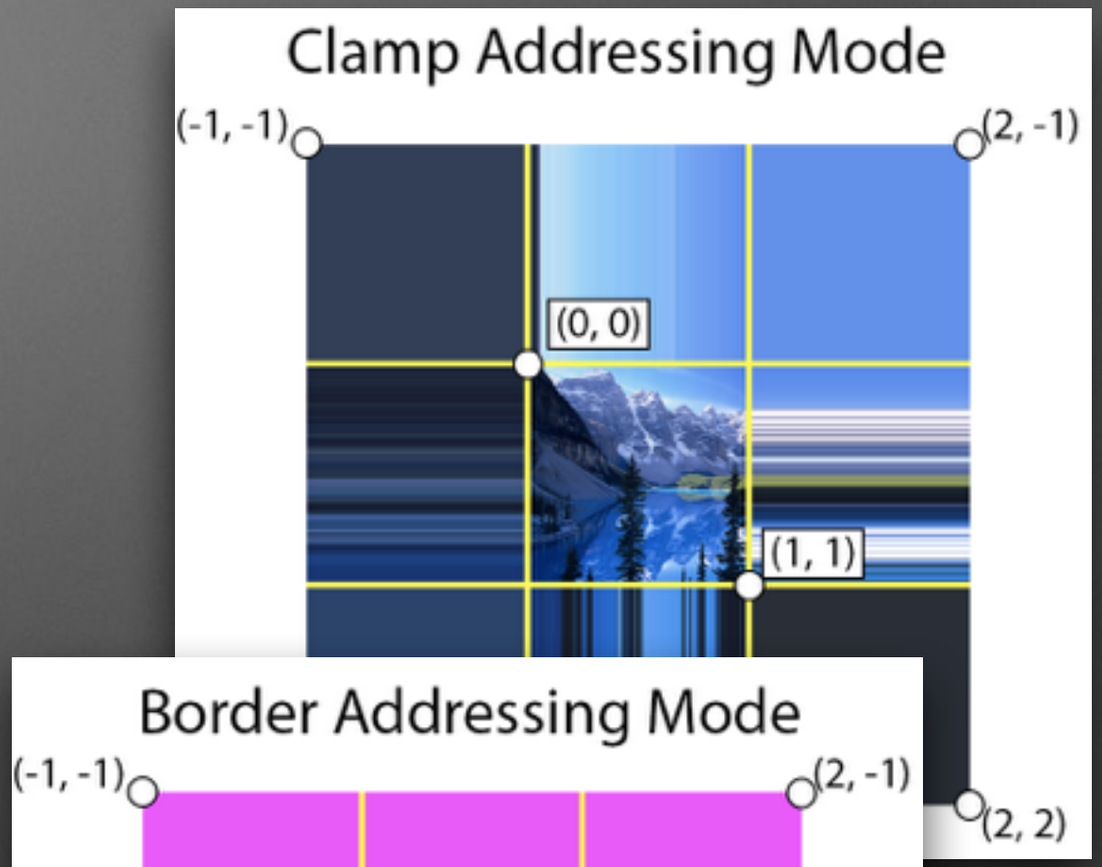
- *wrap:* texture is repeated in both uv directions
- *mirror:* like wrap mode, texture mirrored about v-axis for odd integer multiples of u; and about the u-axis for odd integer multiples of v

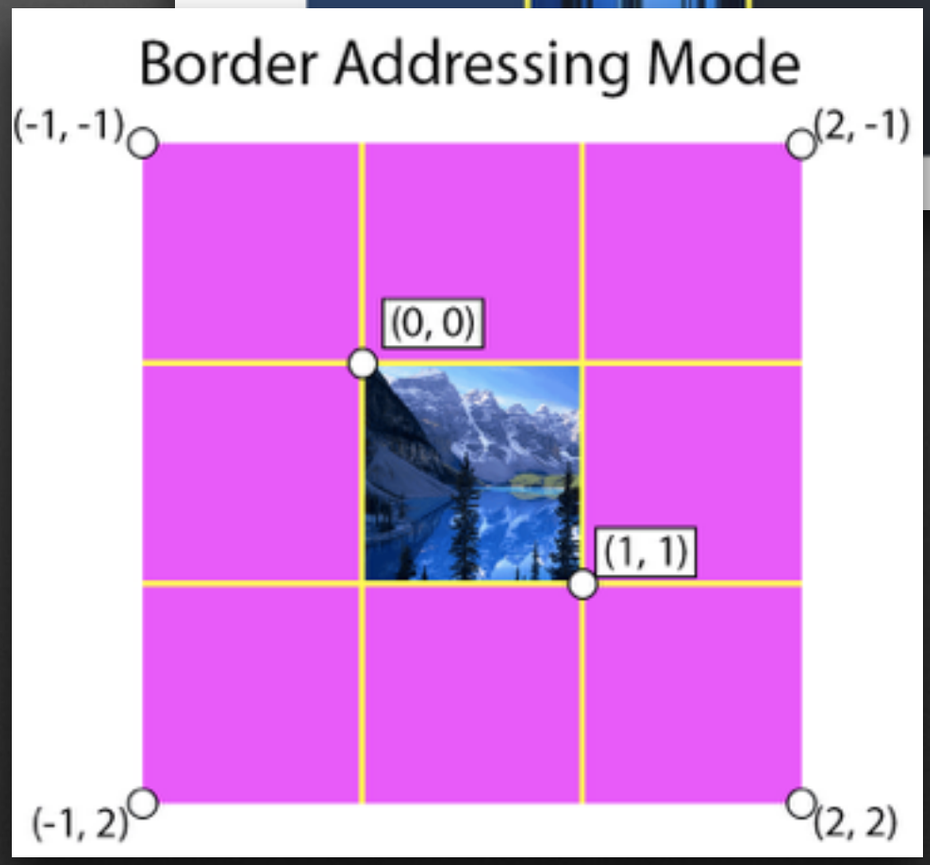




Texture Addressing Mode

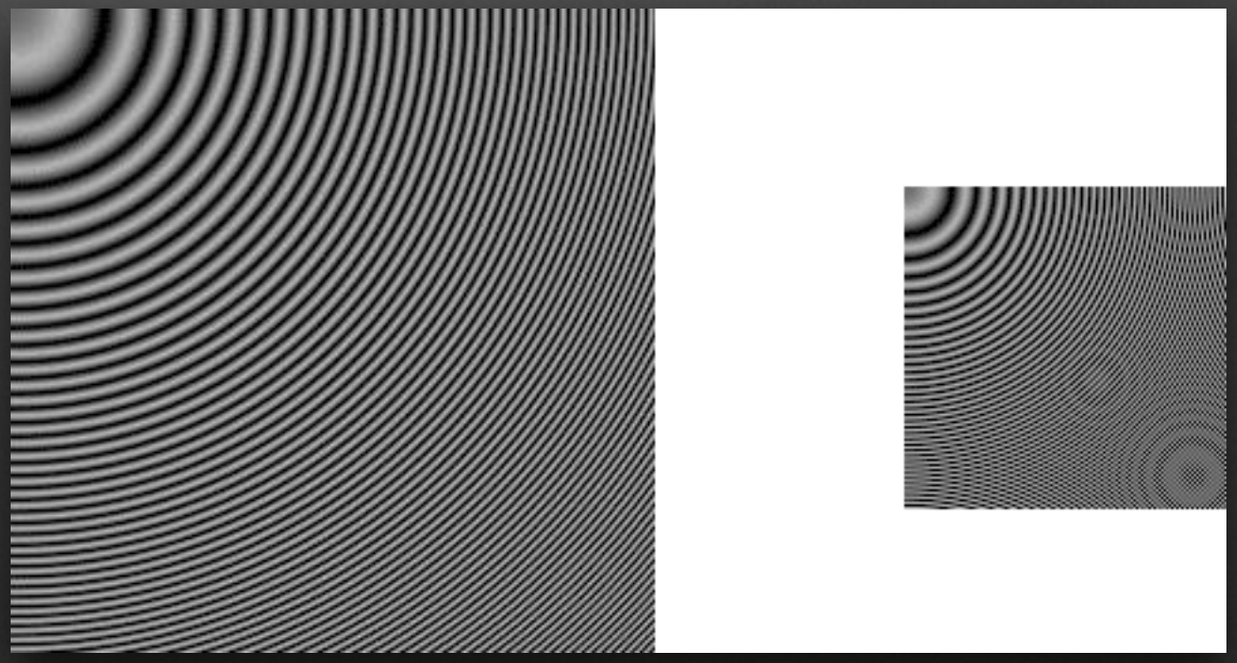
- ***Clamp:*** colors of the texels around the outer edges of the texture are simply extended
- ***Border color:*** same as clamp, except an arbitrary user-specified color is used for the region outside the range





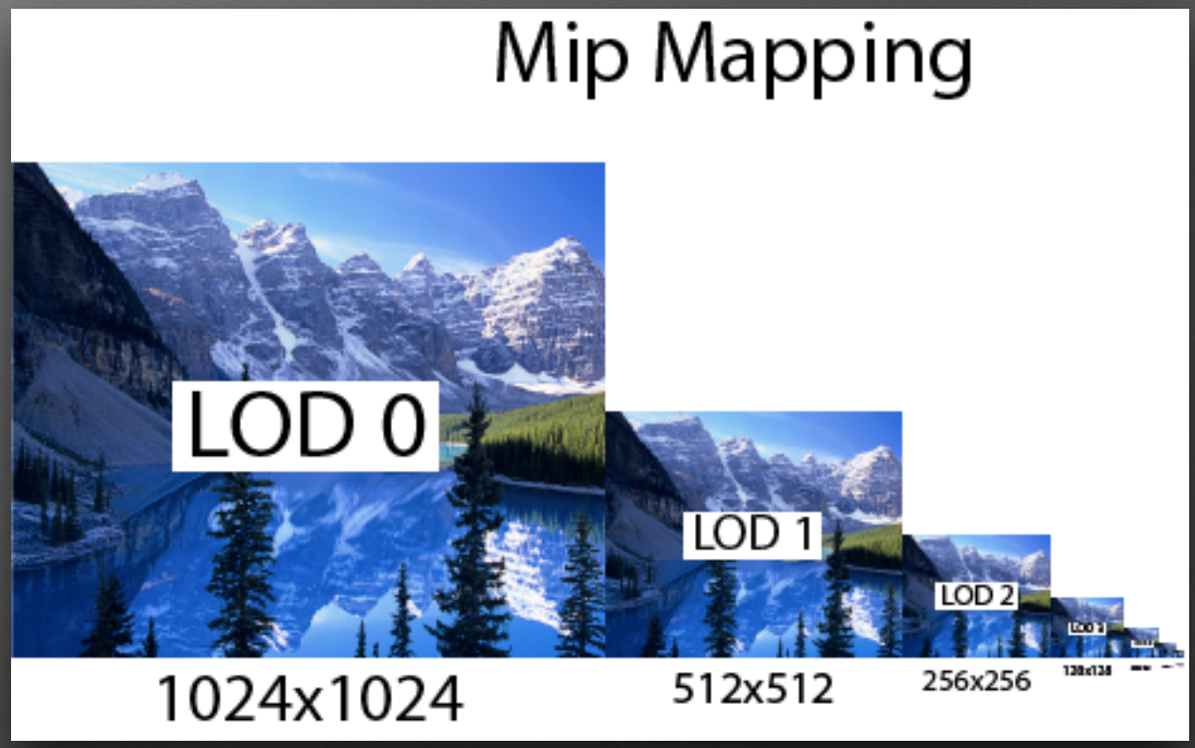
Texel density

- texel density : the ratio of texels to pixels
- texel density affect the visual quality



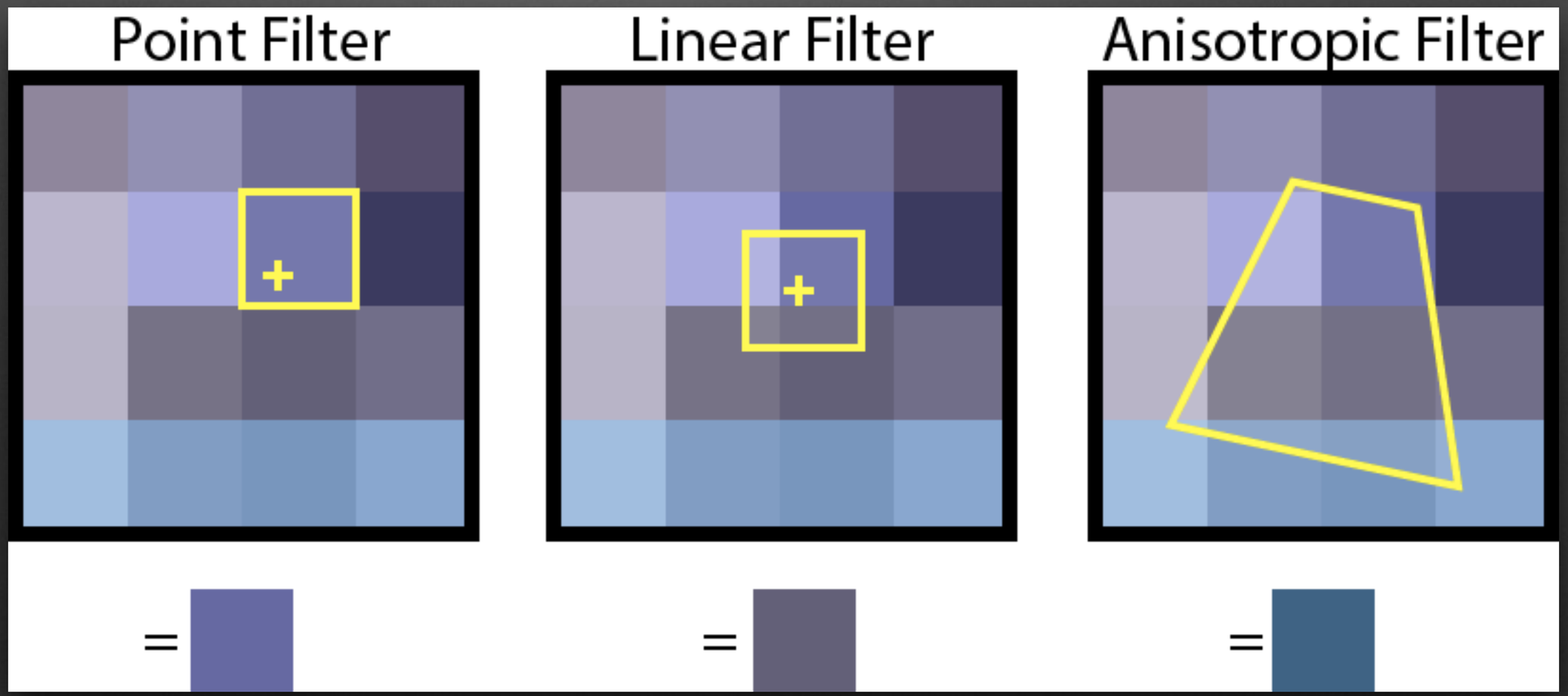
Mipmapping

- a technique to maintain texel density around 1
- pre-filter texture at multiple resolutions

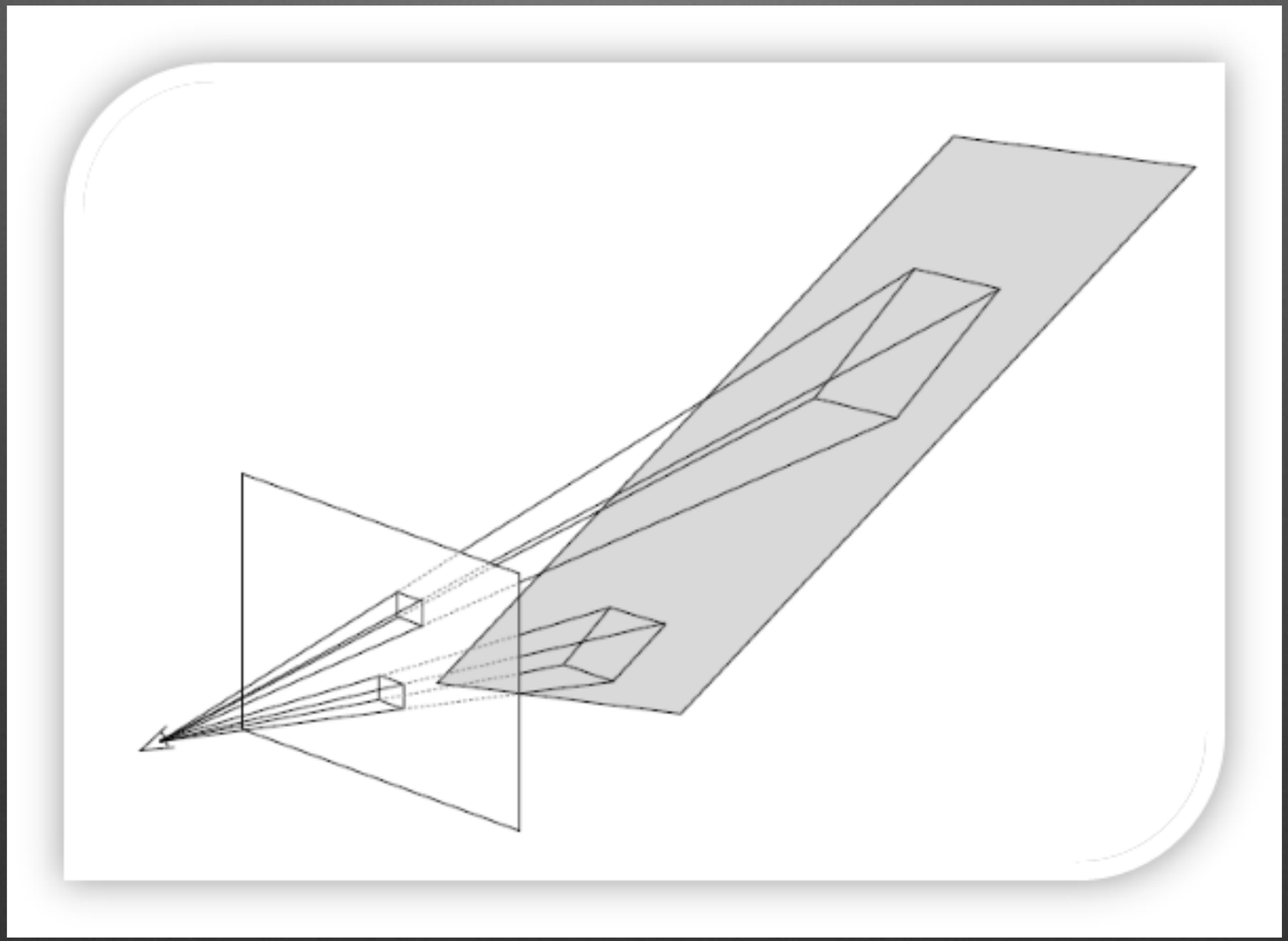


Texture Filtering

- graphic hardware have to sample more than one texels to produce the final result



Anisotropic sampling



Material

- a complete description of the visual properties of a mesh
- including:
    - textures
    - shader and its input parameters
    - other parameters to control the rendering pipeline
- exclude vertex attributes, which is belonged to mesh
- mesh-material pair contains all the information we need to render the object

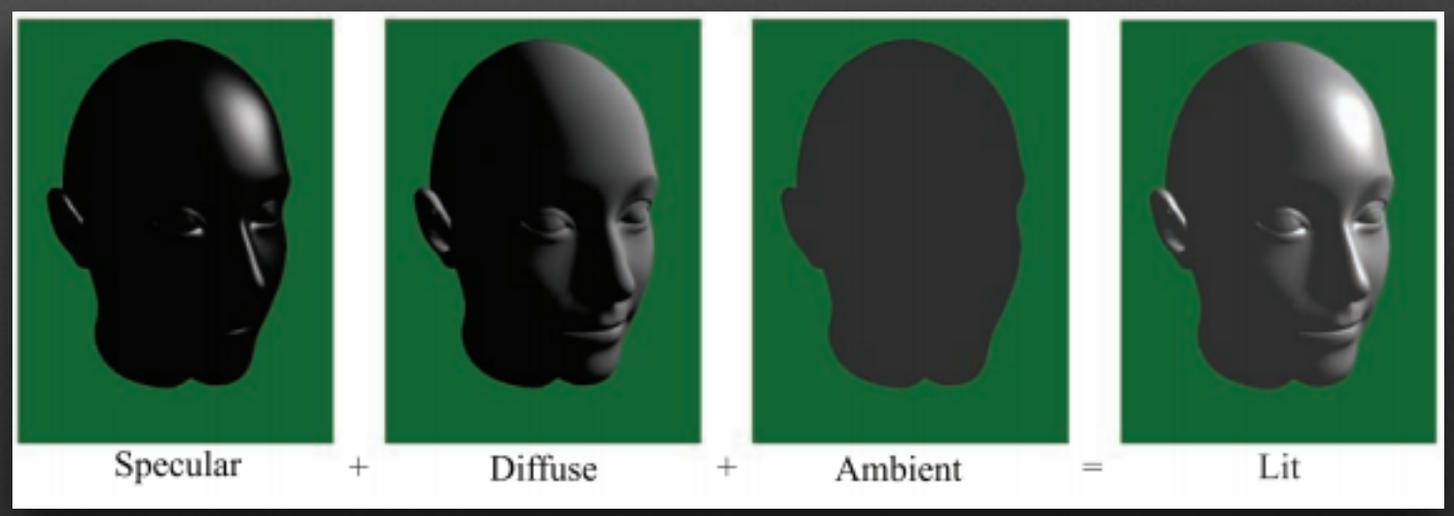
Lights

Light transport model

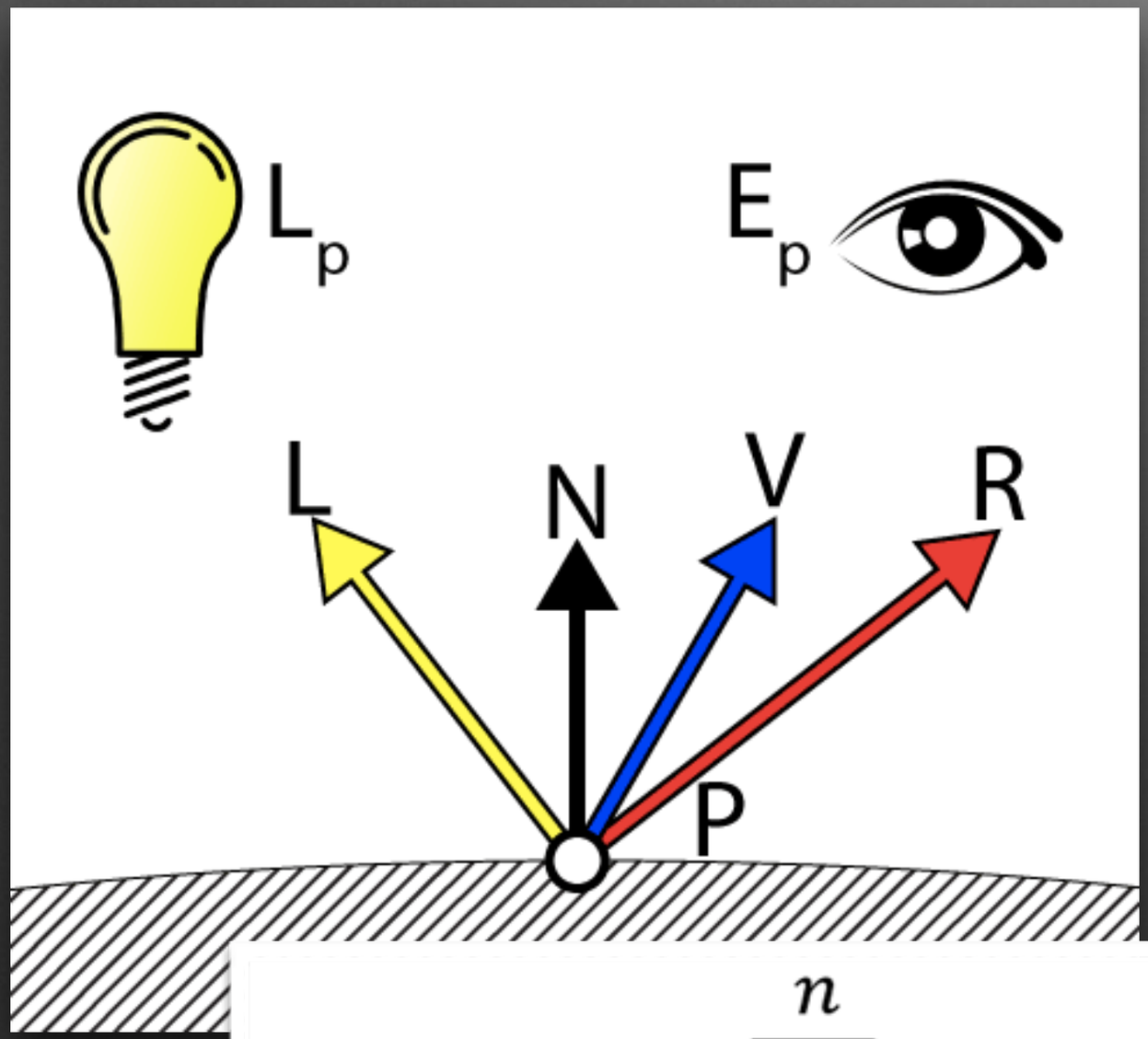
- local illumination model
- global illumination model

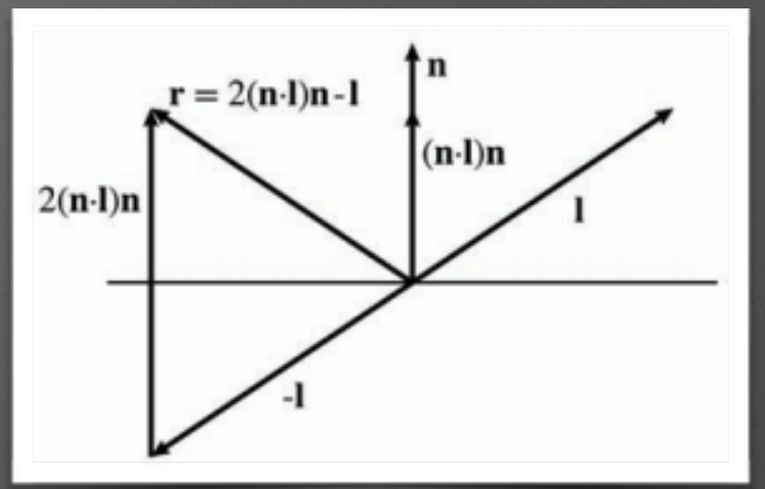
The Phong lighting model

- the most common local lighting model
- light reflected from surface is the sum of three distinct terms:
  - ambient term
  - diffuse term
  - specular term



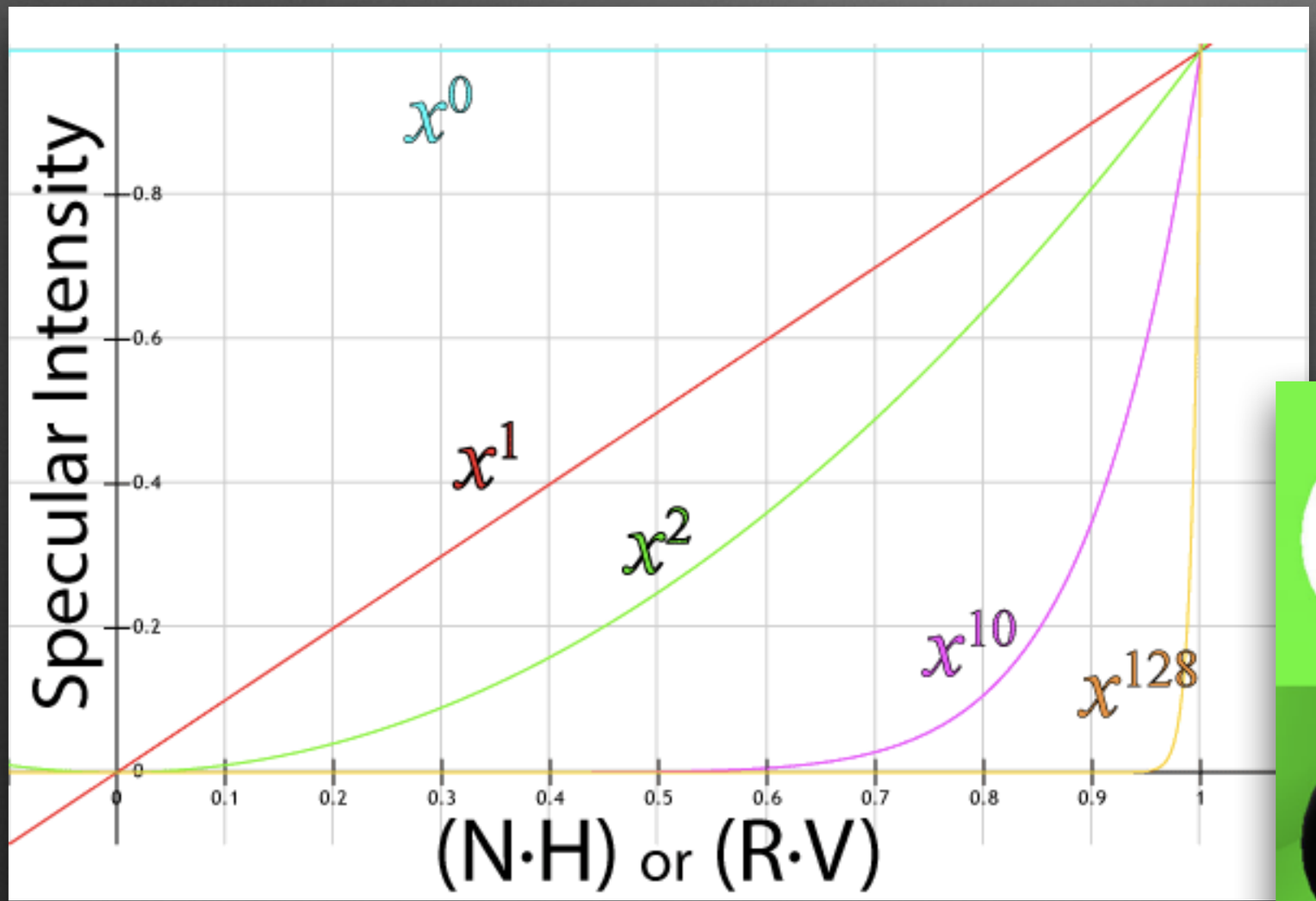
The Phong lighting model

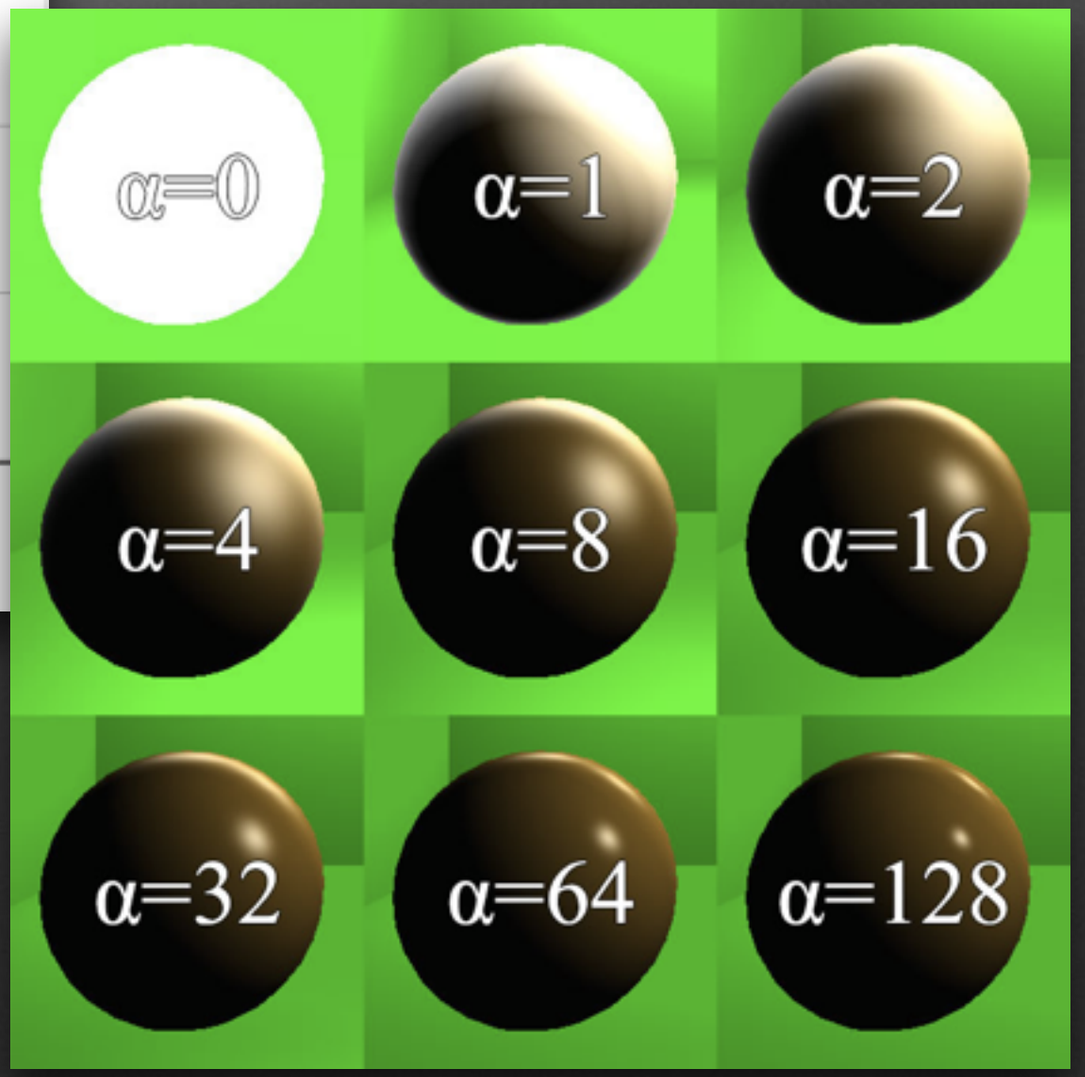




$$I = k_A A + \sum_{i}^{n} [k_D (N \cdot L_i) + k_s (R_i \cdot V)^a] C_i$$

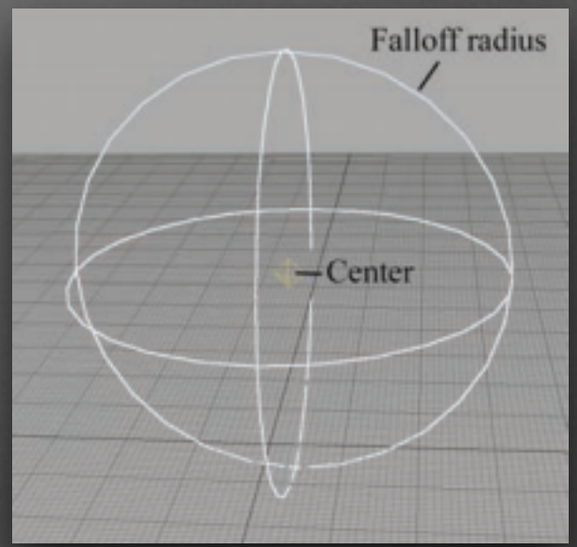
different specular value

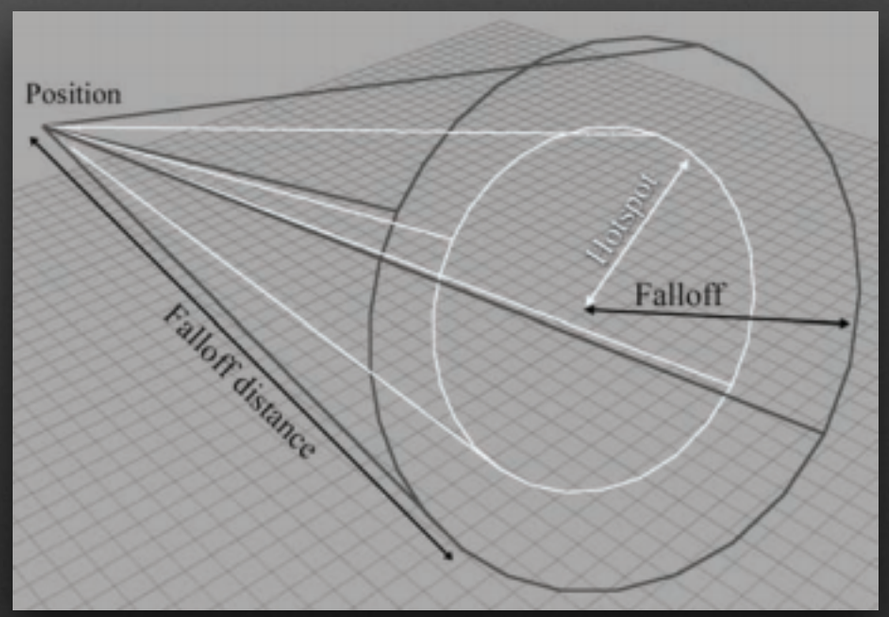




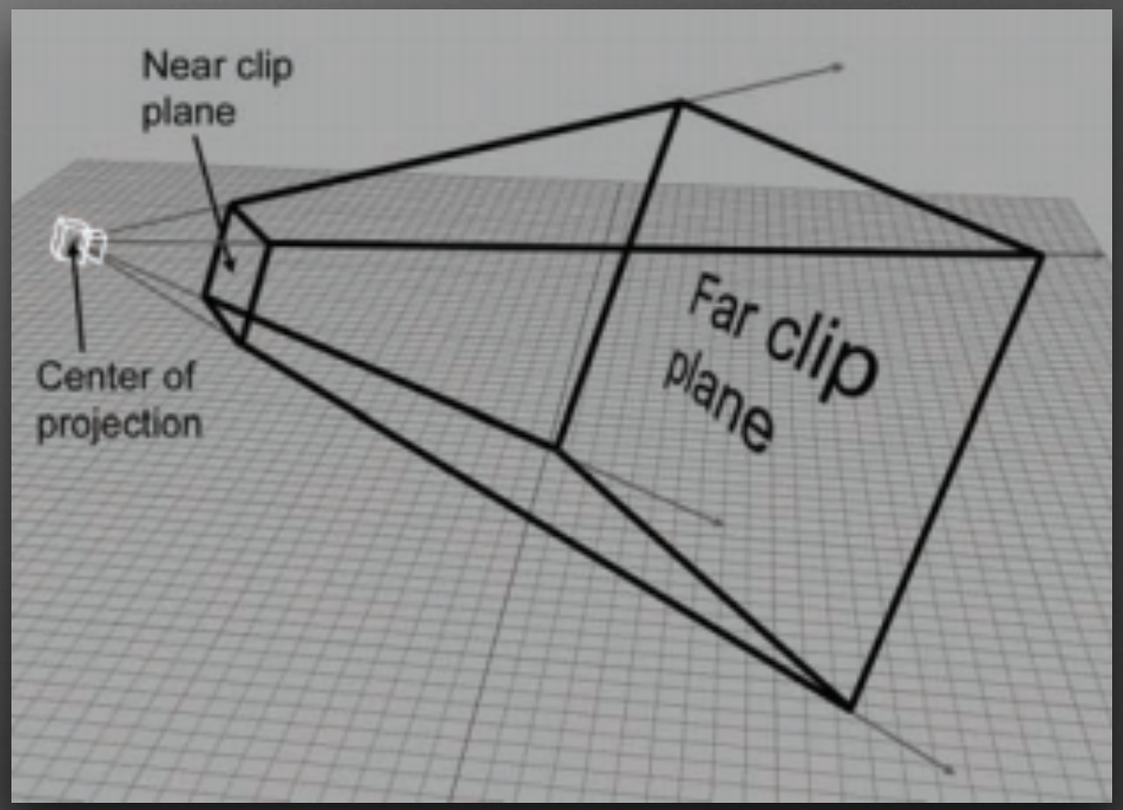
Light sources model

- Directional Lights
- Point Lights
- Spot Lights

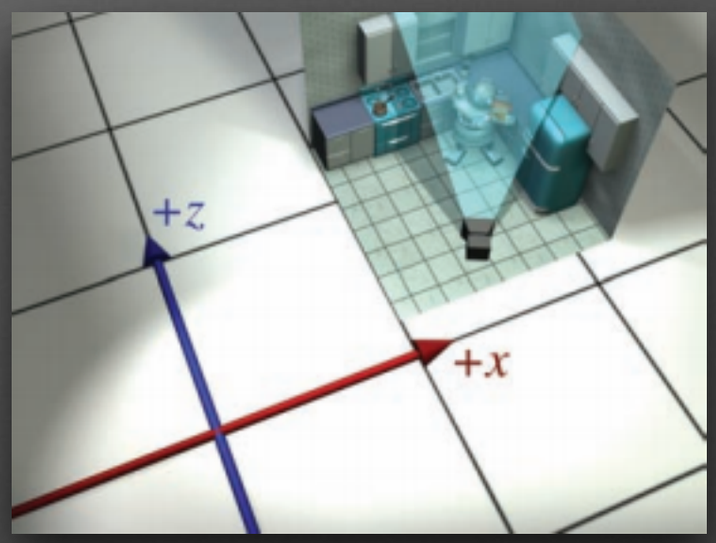




The Virtual Camera



Camera Space



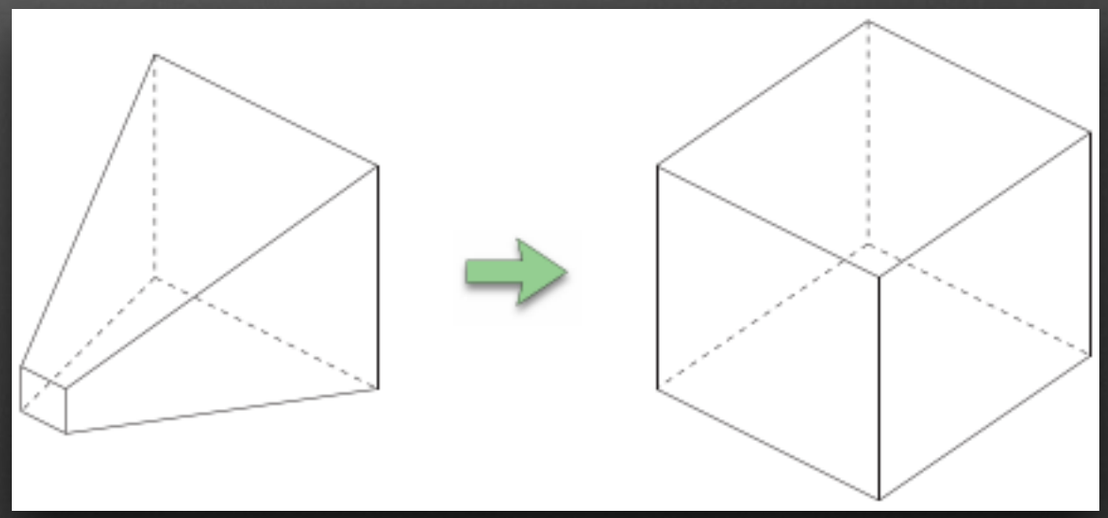
- the origin is the focal point of the virtual camera
- the looking direction is z-axis
- y-axis is the up direction of the camera
- view matrix:

$$M_{V\to w} = \begin{bmatrix} \boldsymbol{i}_V & 0 \\ \boldsymbol{j}_V & 0 \\ \boldsymbol{k}_V & 0 \\ \boldsymbol{t}_V & 1 \end{bmatrix}$$

$$M_{w\to v} = (M_{v\to w})^{-1} = M_{view}$$

Projection

- transform vertices from camera space to clip space
- prepare for clipping
- prepare for projection



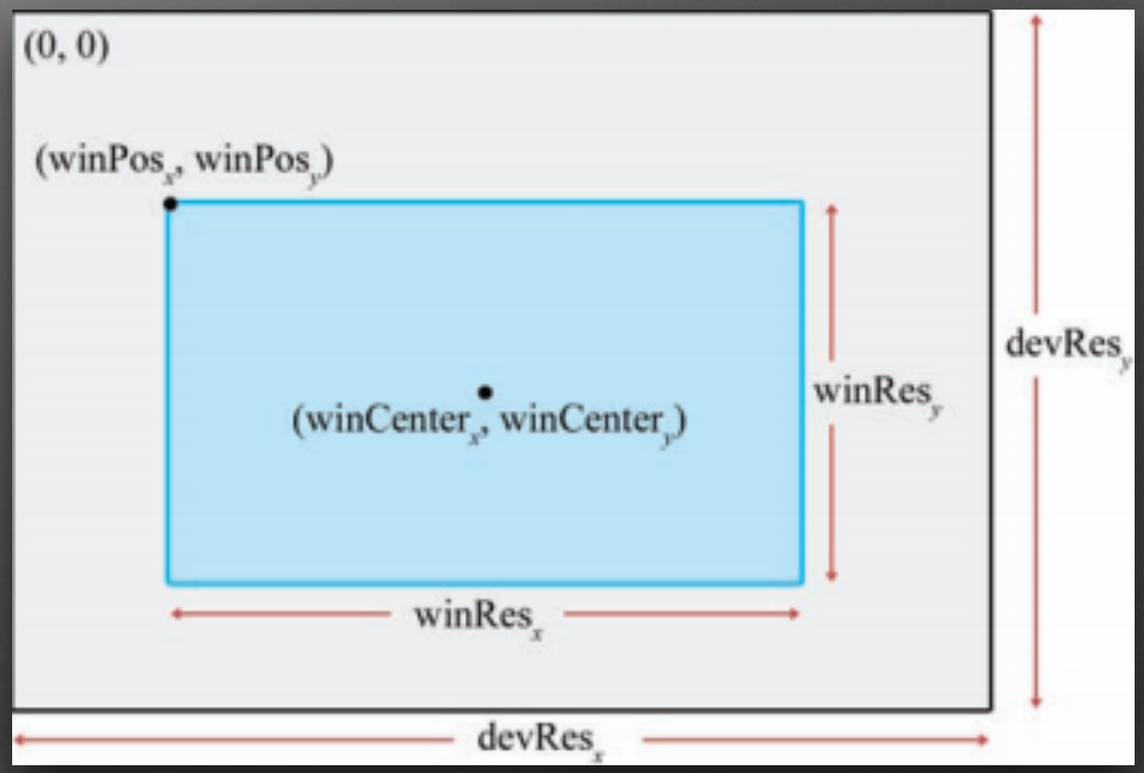
Projection Matrix

$$M_{proj} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}$$

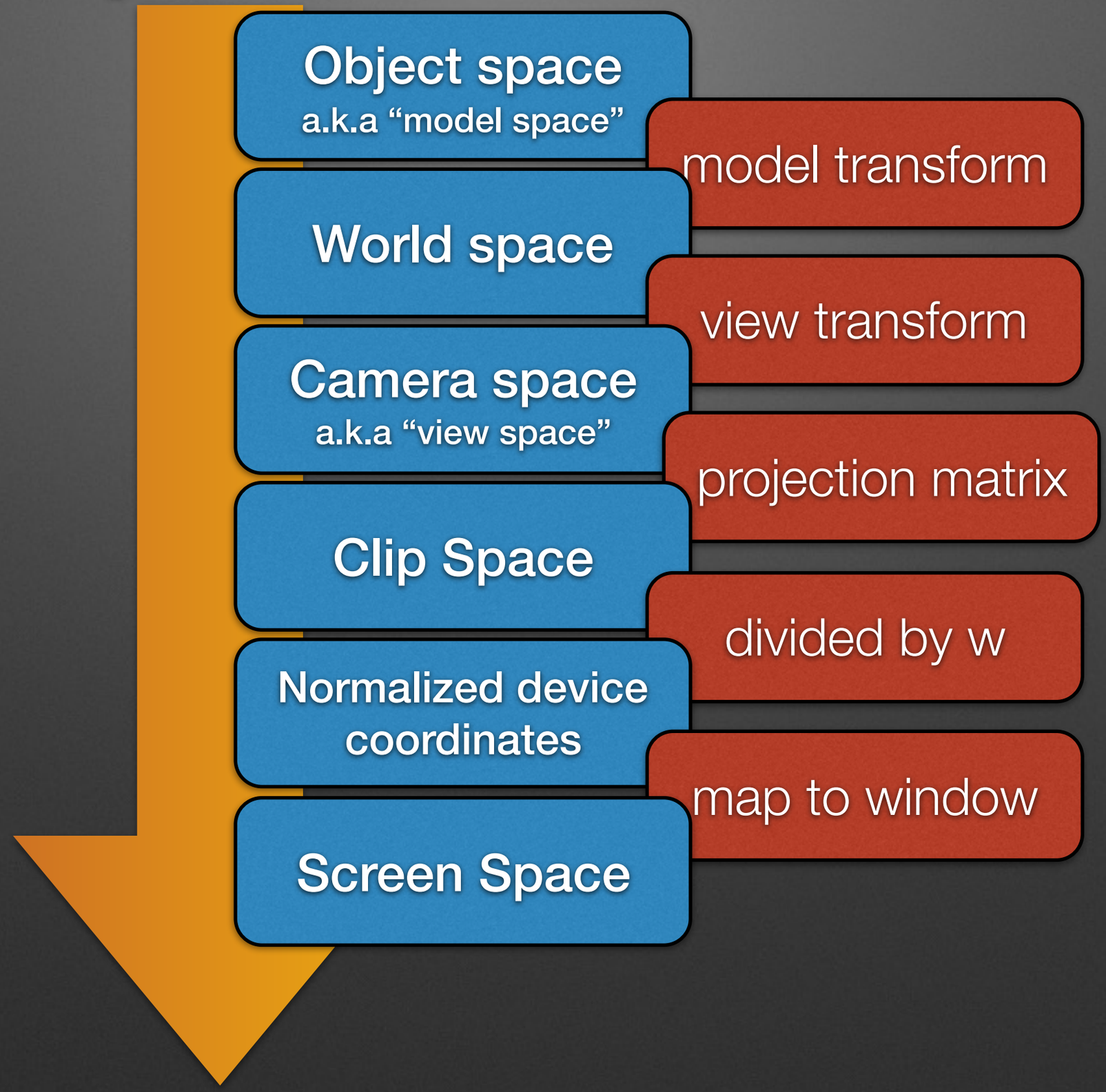
Screen Space

- a two-dimensional coordinate system whose axes are measured in terms of screen pixels

- just scale-and-shift the (x, y) coordinates from clip space, this is call *screen mapping*

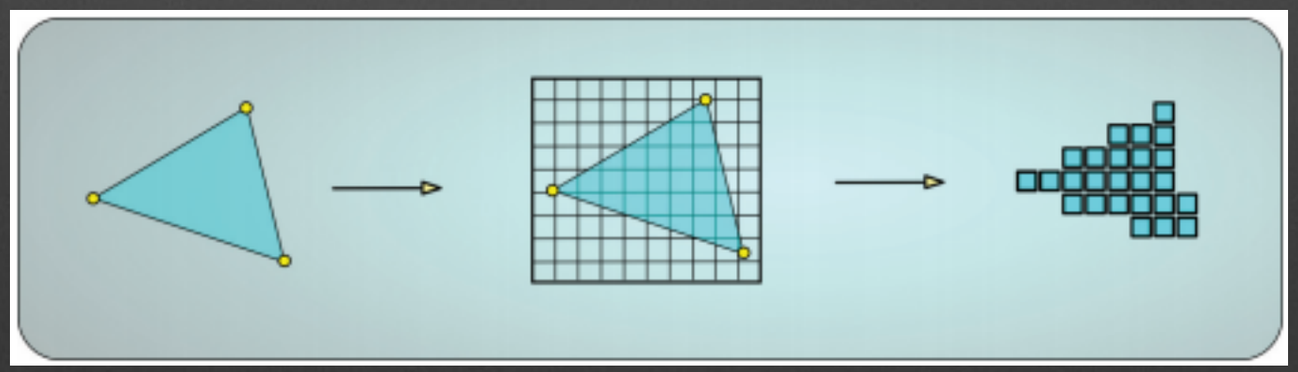


Space Transforms



Rasterization

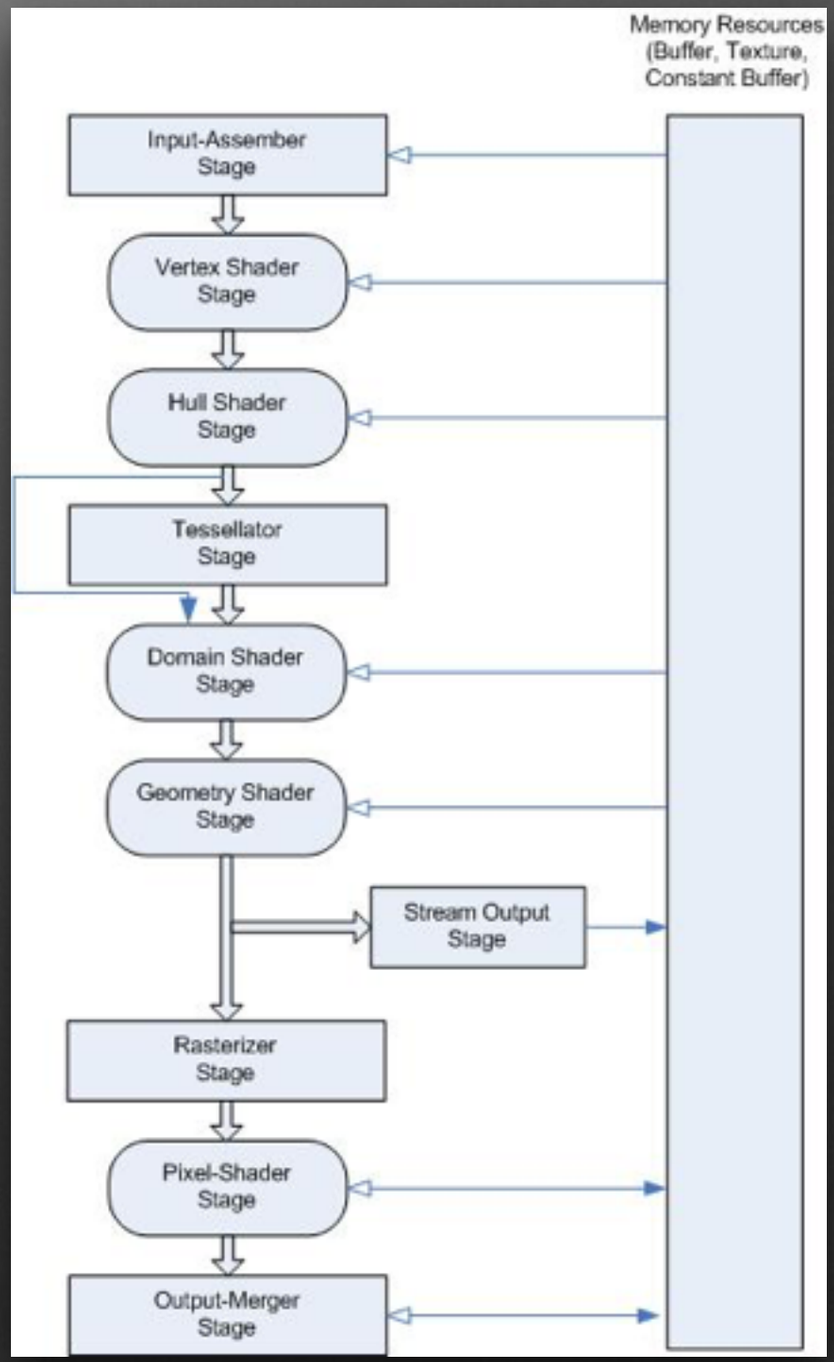
- a process to fill the pixels a triangle overlaps on-screen
- a triangle's surface is broken into pieces called *fragment*



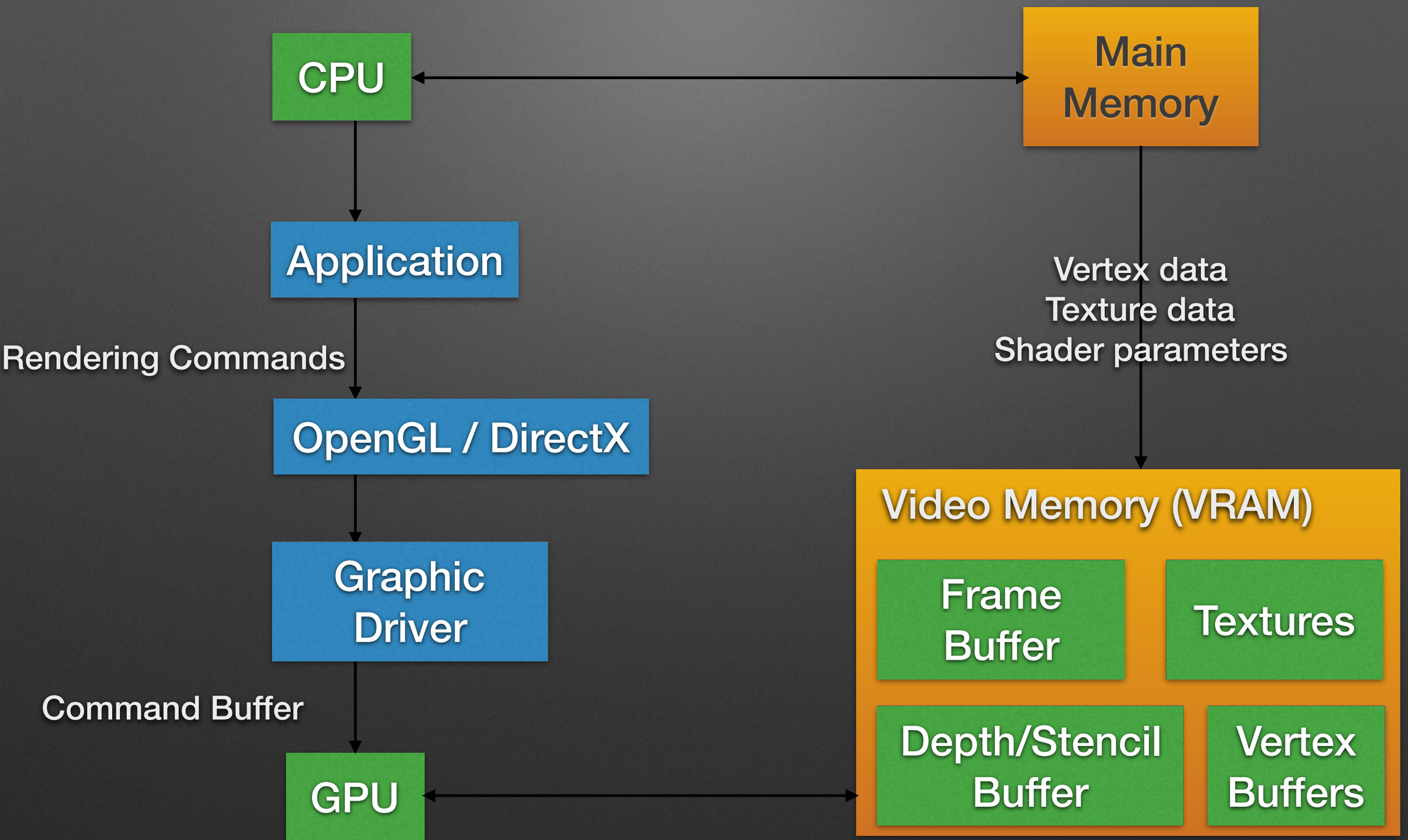
Rendering Pipeline

- In real-world, the rendering steps described before are implemented using a software/hardware architecture known as *rendering pipeline*

- a mechanism used to process memory resources into a rendered image



Rendering Pipeline



Pipeline Stages

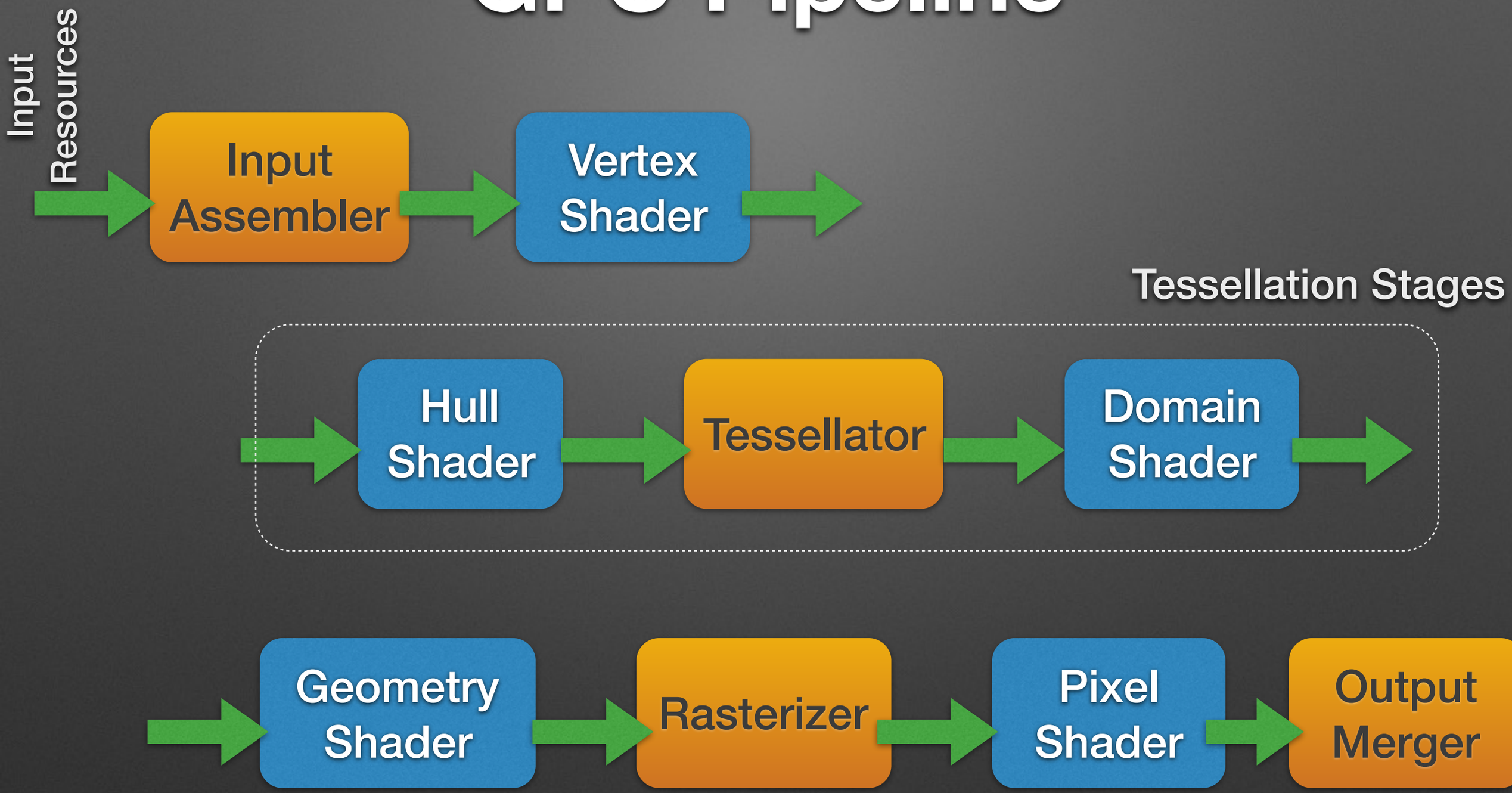
- pipeline itself is made up of a number of small logical units, called *pipeline stages*

- the task of the developer is to properly configure each stage of the pipeline to obtain the desired result

- Fixed Pipeline Stages

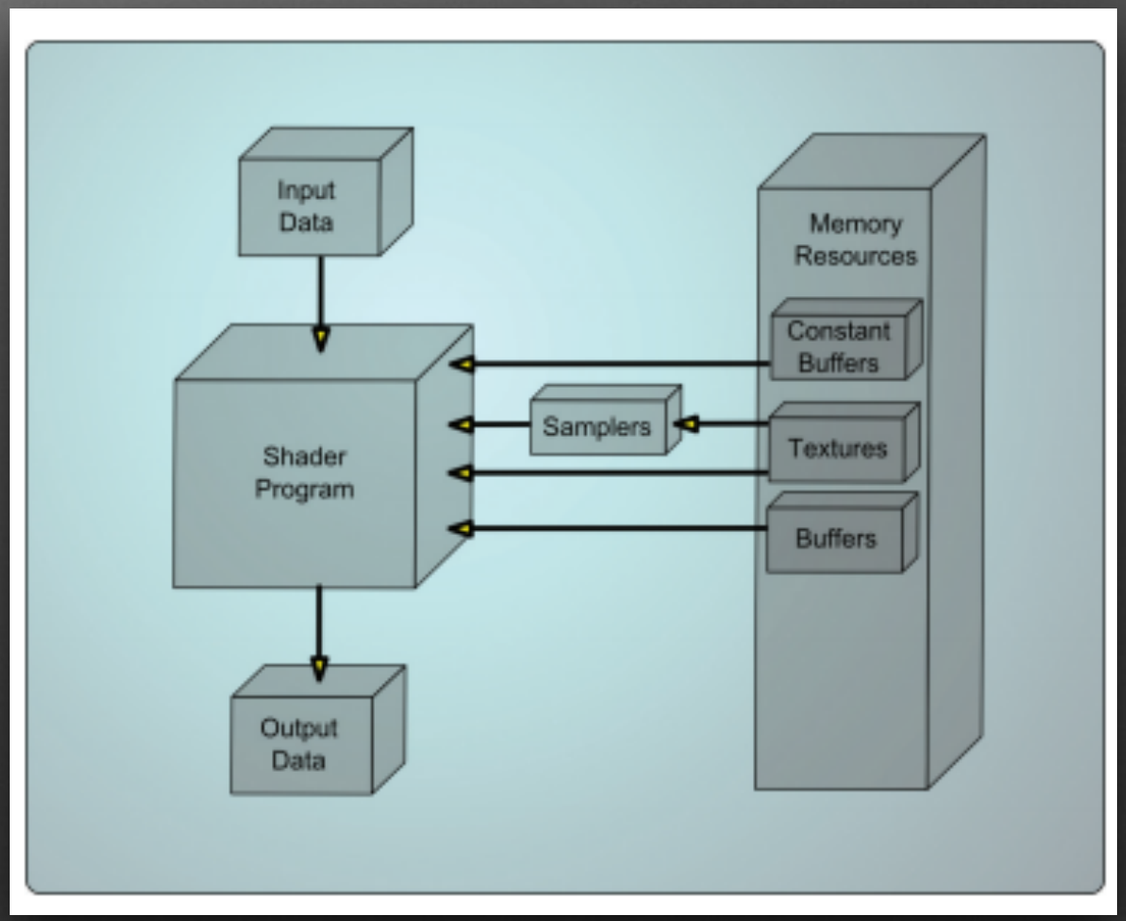
- Programmable Pipeline Stages

GPU Pipeline



Common Shader Core

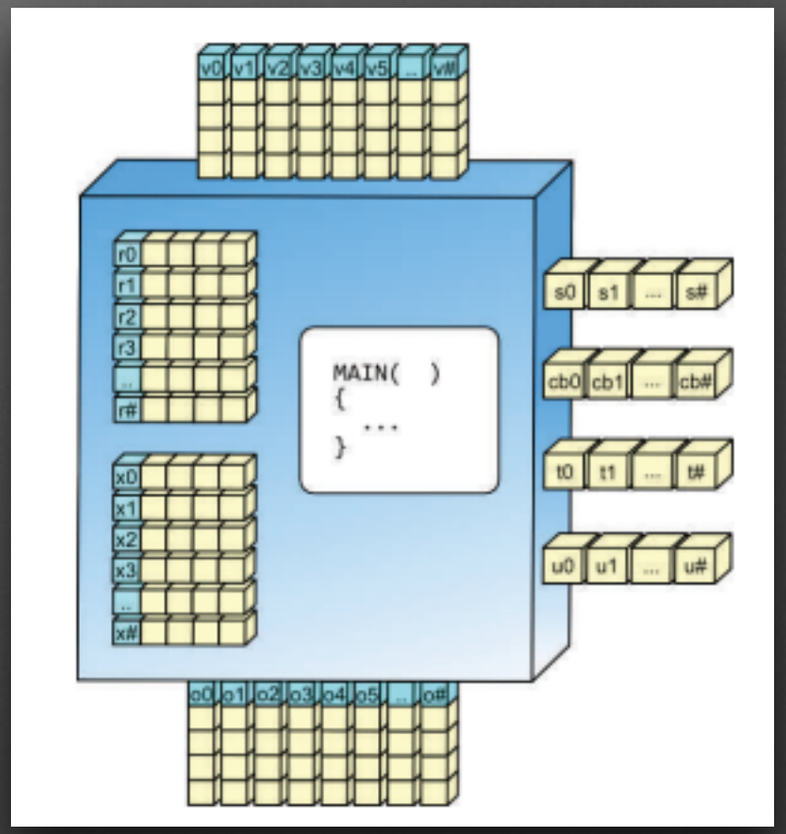
- All of the programmable shader stages are built upon a common base of functionality, call *common shader core.*



Shader Core Architecture

- Shader program must be compiled into a vector-register-based assembly language before being used by *rendering pipeline*

- assembly version of the common shader core



stage-to-stage communication

- each shader defines its required inputs and outputs for its shader program as *attributes*;

- The input attributes are the input parameters to the shader program function, the return value defines its output attributes;

- attributes are implemented with the input & output registers;

- Attributes are labeled with text-based identifier in shader program call *semantics*

stage-to-stage communication

```
struct VS_INPUT
{
    float3 position : POSITION;
    float2 color    : TEXCOORDS;
};

struct VS_OUTPUT
{
    float4 position : SV_POSITION;
    float4 color : COLOR
};

VS_OUTPUT VSMAIN(in VS_INPUT v)
{
    // Perform vertex processing here ...
}

float4 PSMAIN(in VS_OUTPUT input) : SV_Target
{
    // Perform pixel processing here ...
}
```

A brief history of GPU

- At first, everything is done on the CPU
- then comes the 3Dfx's Voodoo, only for rasterization
- then fixed-function pipeline (hardware T&L)
- then we have shaders(VS, PS)
- more shaders(GS, HS, DS)
- GPGPU

Input Assembler

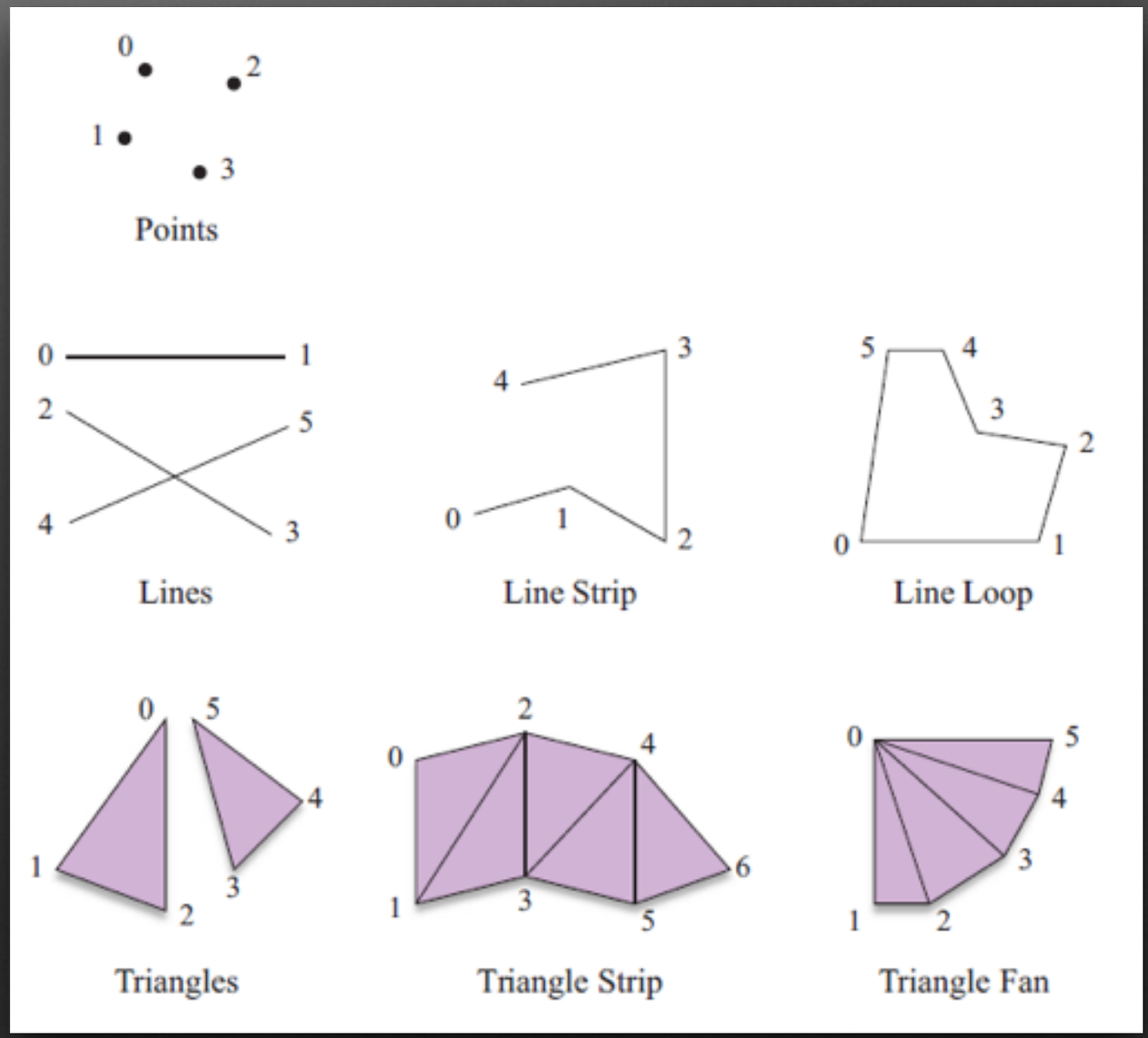
- first stop in the rendering pipeline
- responsible for putting together all the vertices that will be processed by the pipeline
- determine how vertices are connected to one another by configuration of topology

IA Input

- Vertex Buffer
- Index Buffer

IA Configuration

- Input Layout
- Primitive Topology



IA Output

- user defined semantic attributes:
  - position
  - normal
  - texcoords
  - etc.
- some system level semantic attributes can also be produced

Vertex Shader

- first programmable shader stage;
- invoked for each vertex in the vertex stream produces by IA;
- no access to primitive stream

VS Input

- Vertex stream from IA;
- Some system level semantic attributes

VS Configuration

- Shader Program
- Shader Program Pamameters
- Shader Resources

VS Processing

- Geometric Manipulation
- Vertex Lighting
- Control Point Processing

VS Output

- Depending on what stages are followed:
    - Rasterizer
        - *Clip space position*
    - Geometry Shader
        - *Clip space position (optional)*
    - Tessellation
        - *Clip space position (optional)*
        - *control points*

Rendering Pipeline II

Ju Heqi （Autodesk Inc.）

Geometry Shader

- the final pipeline stage that can manipulate geometry;
- has unique capabilities not found in other stages;
- the heart of the GS operation is the streaming model.

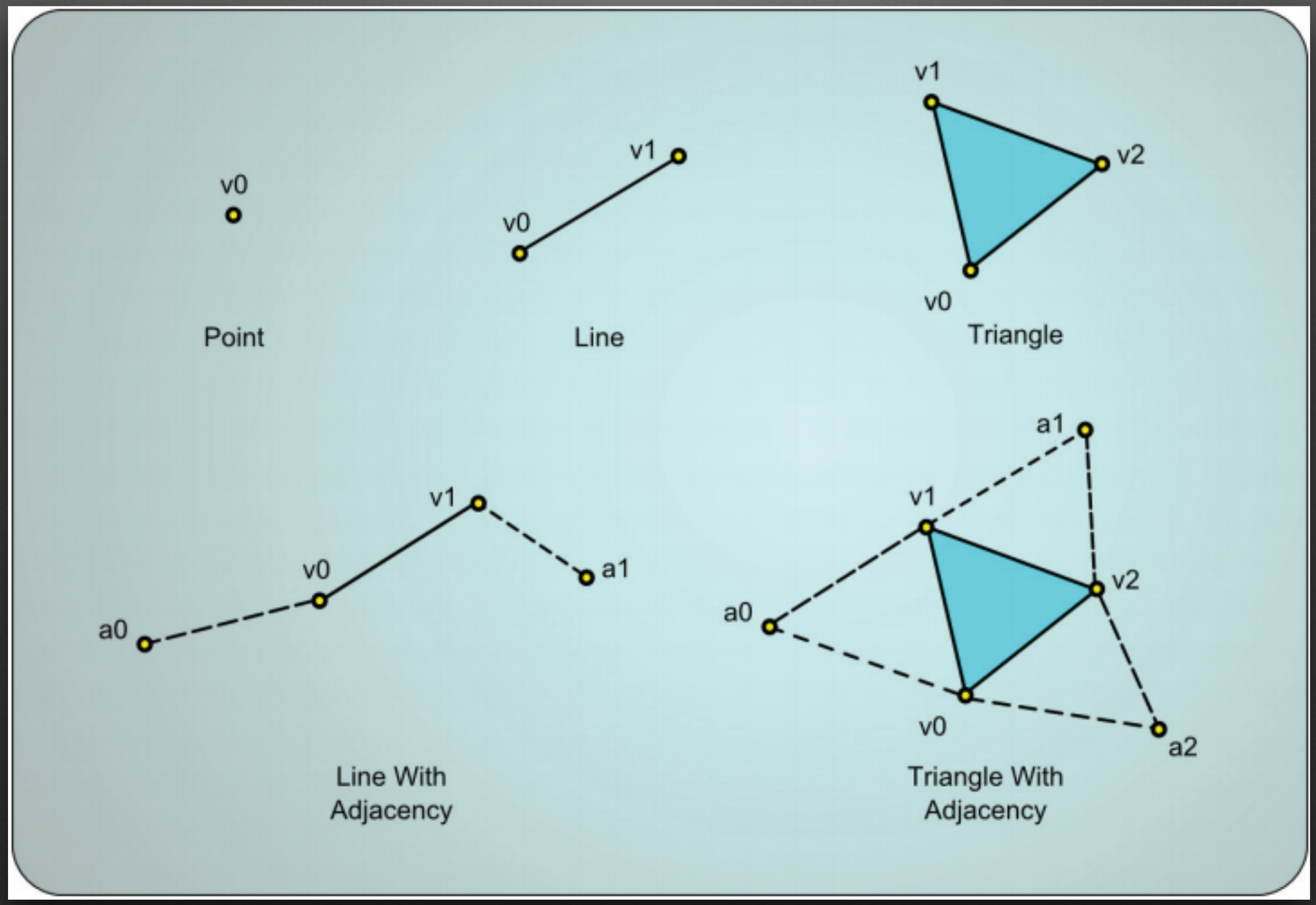
GS Input

- GS operates on complete primitives composed of an array of vertices;
- Array size is decided by the primitive type;

Sample GS code

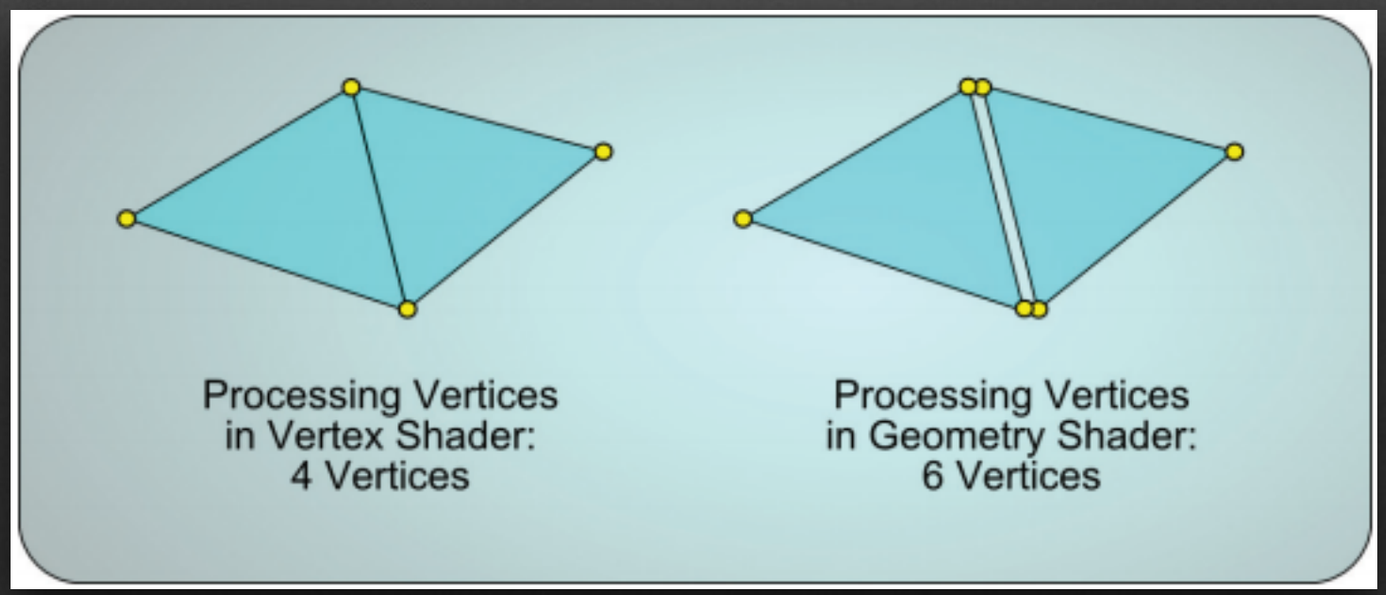
```
[instance(4)]
[maxvertexcount(3)]
void GSScene(triangleadj GSSceneIn input[6],
             inout TriangleStream<PSSceneIn> OutputStream)
{
    PSSceneIn output = (PSSceneIn)0;

    for (uint i = 0; i < 6; i += 2)
    {
        output.Pos  = input[i].Pos;
        output.Norm = input[i].Norm;
        output.Tex  = input[i].Tex;

        OutputStream.Append(output);
    }

    OutputStream.RestartStrip();
}
```

Primitives available to GS



GS Input

- two different possible pipeline setup that can send primitive data to GS, depends on if tessellation is active;
- Performance implication in GS.



GS Configuration

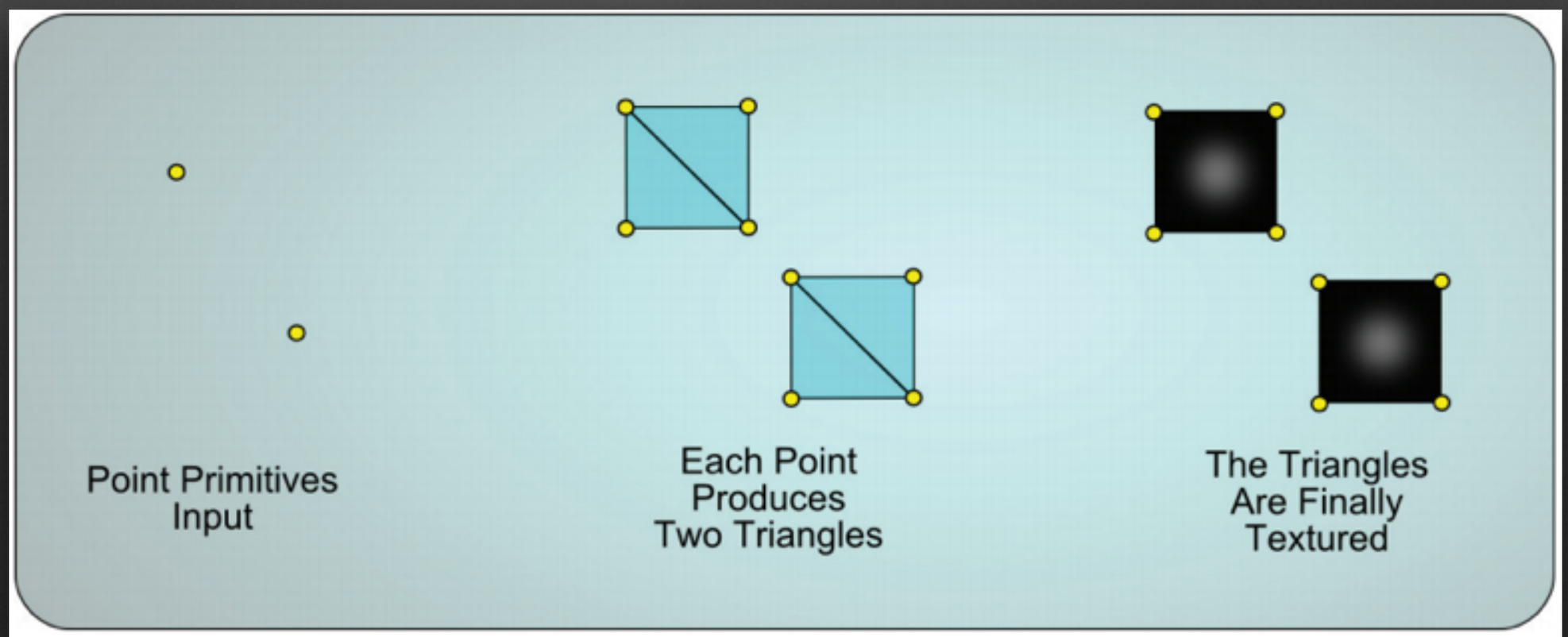
- Common shader core resource configuration;
- Function Attributes

GS Processing

- GS Process Flow
    - three output stream type available
        - point list
        - line strip
        - triangle strip
- use Append() to generate output primitives
- use RestartStrip() to create detached geometry

GS Processing

- Primitive Manipulations
- Shadow Volumes
- Point Sprites

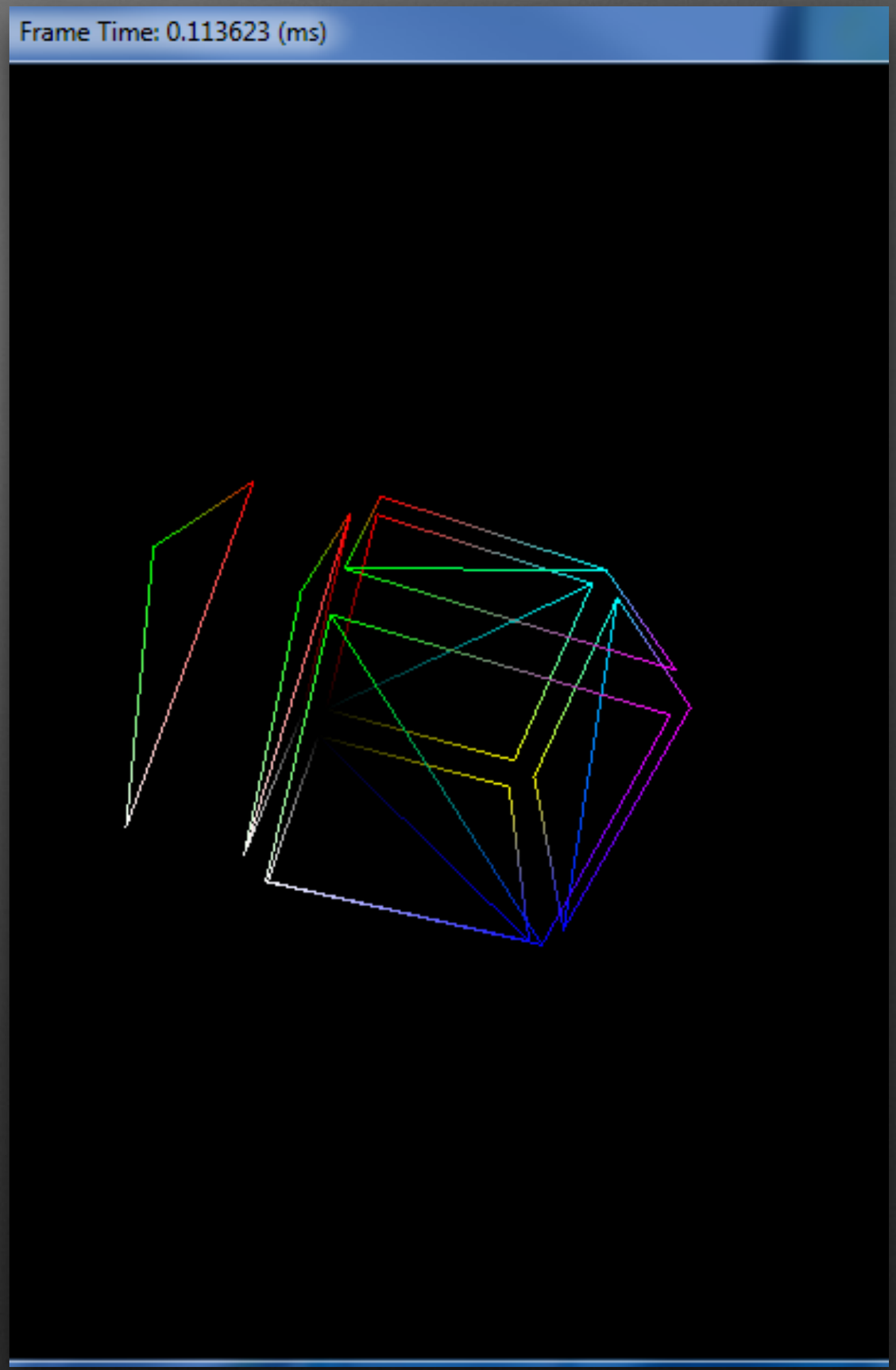


GS Output

- GS pass its results down the pipeline by output streams;
- GS must pass the final clip space position to the rasterizer

Demo

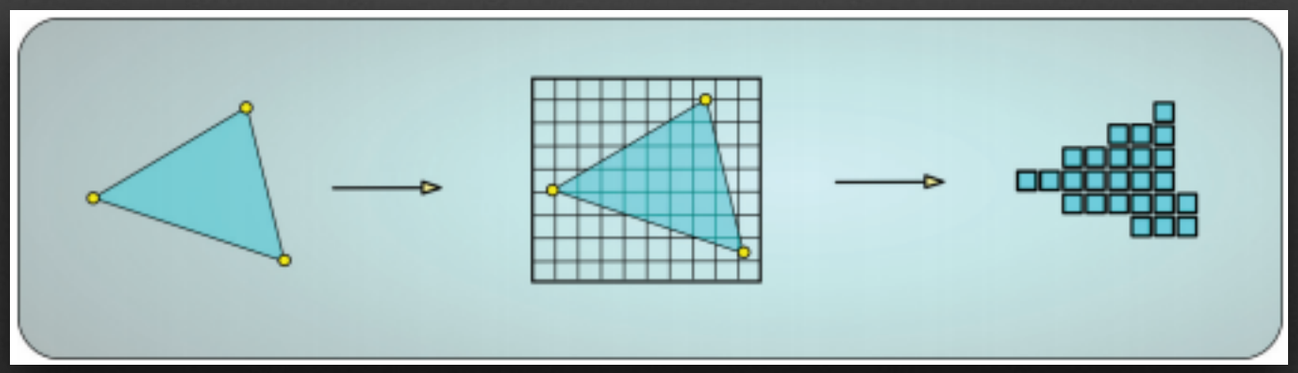
Primitive manipulation with Geometry Shader



Rasterizer

- primary purpose is to convert the geometric data into a regularly sampled representation that can later be applied to a render target, this sampling process is called *rasterization*.

- The result of rasterization is a number of *fragments*



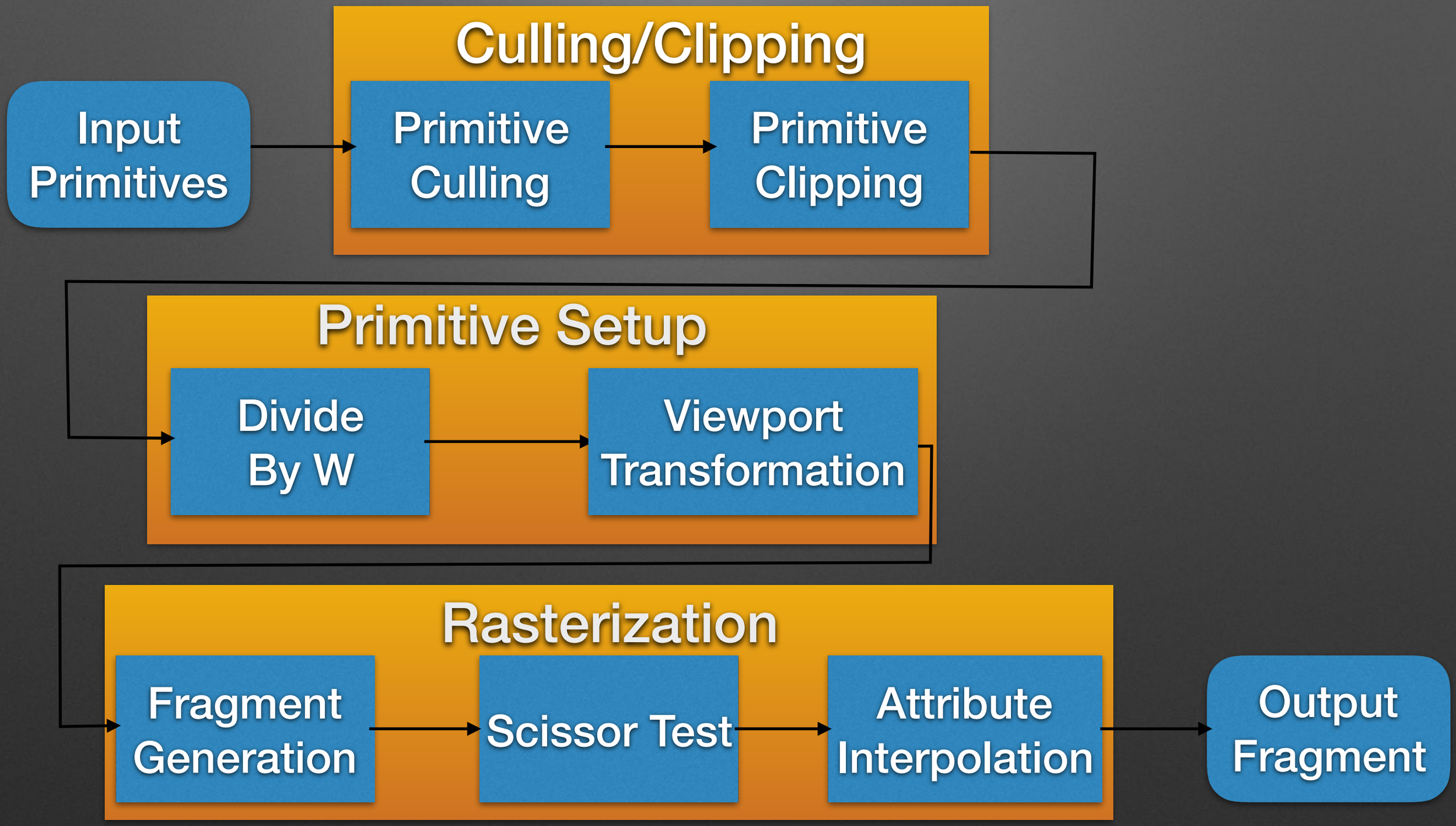
Rasterizer Input

- rasterizer receives individual primitives as its input;
- the primitive data consists of the vertices that define its geometric shape, each vertex must contain one attribute with SV_Position system value semantics;
- clip & cull distance;
- viewport array index;
- additional input vertex attributes

Rasterizer Configuration

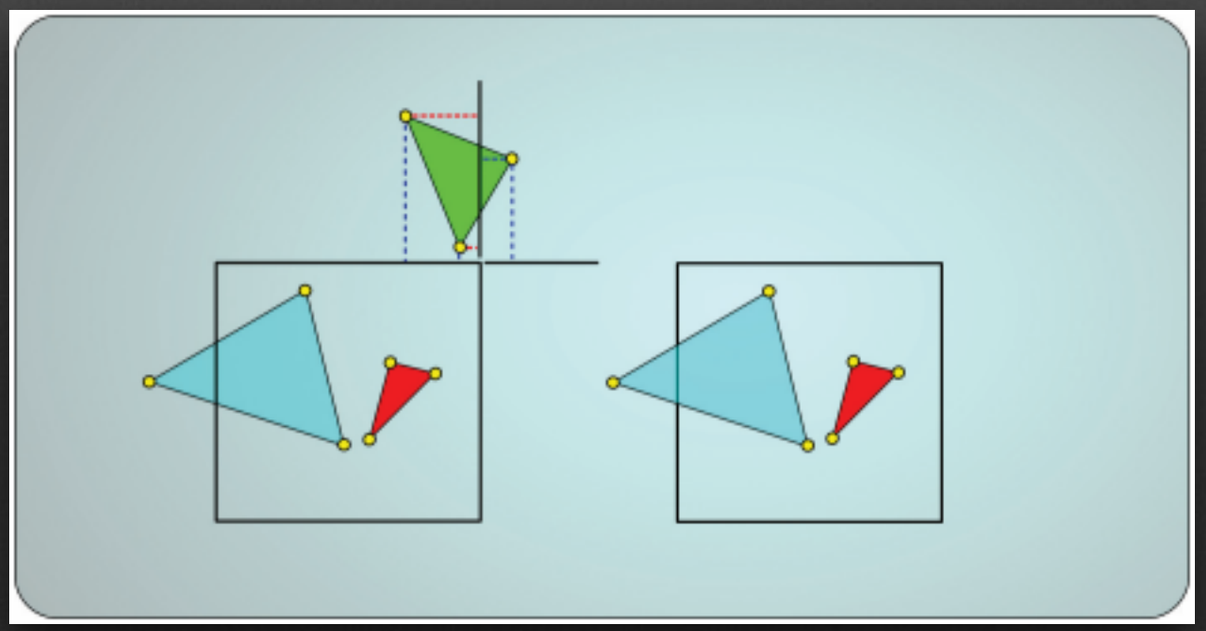
- rasterizer state
    - fill mode;
    - cull mode & front face clockwise;
- viewport state
    - must provide at least one;
    - defines a region the NDC will be mapped;
- scissor rectangle state
    - specify a particular region of the render target for which fragment can be generated.

Rasterizer Processing



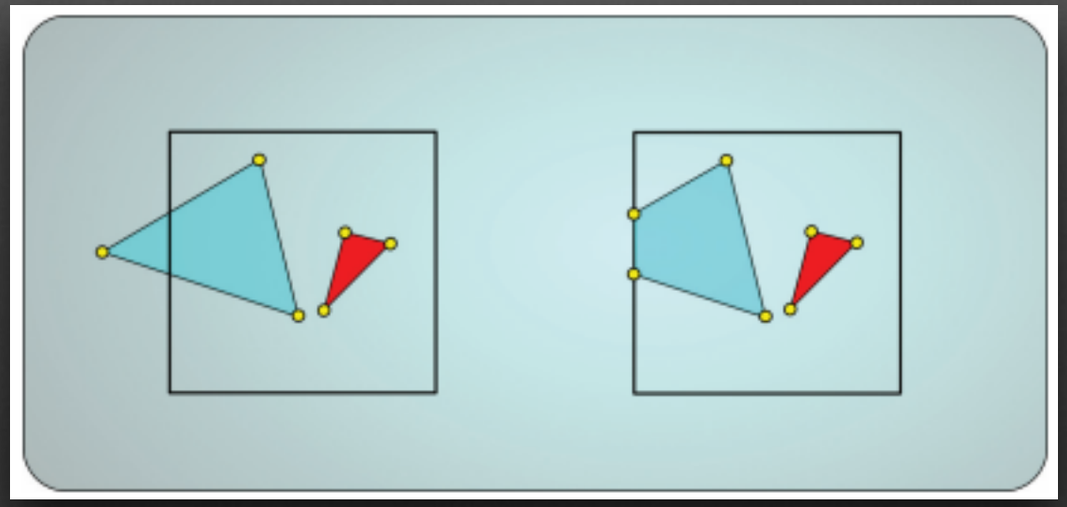
Culling

- back face culling
    - only applied to triangle primitive;
- primitive culling
    - cull primitive that is completely outside the frustum;



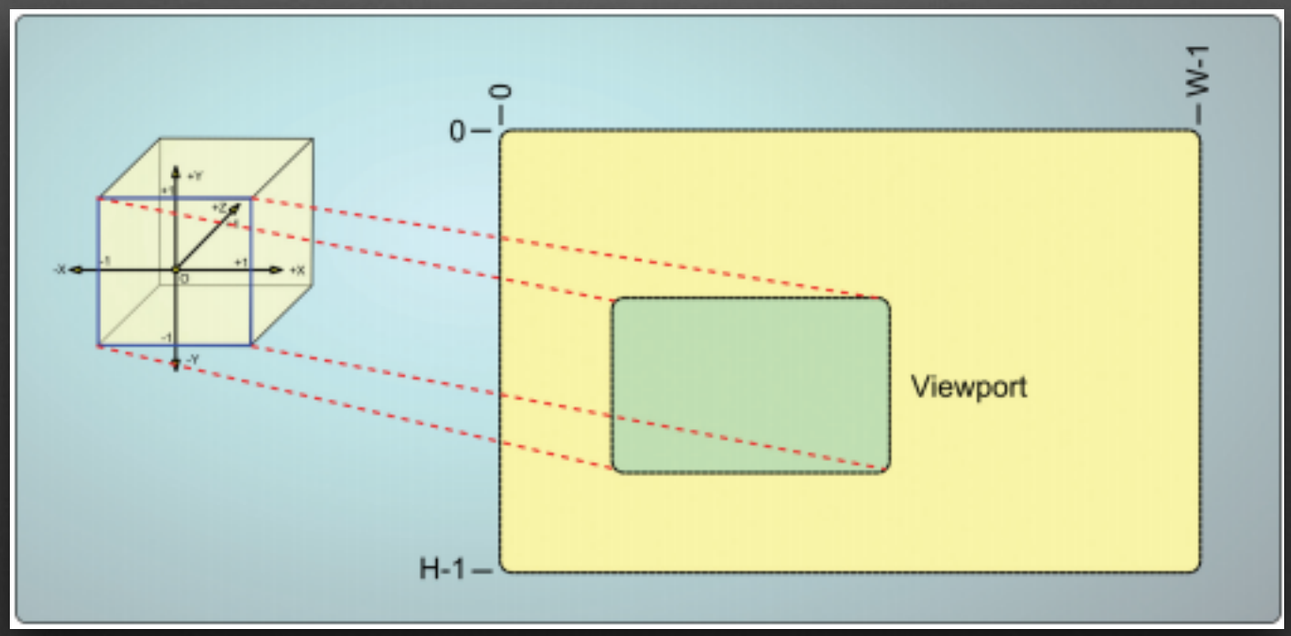
Clipping

- test primitives that survived the culling test to see if it is located fully inside of the *frustum*;



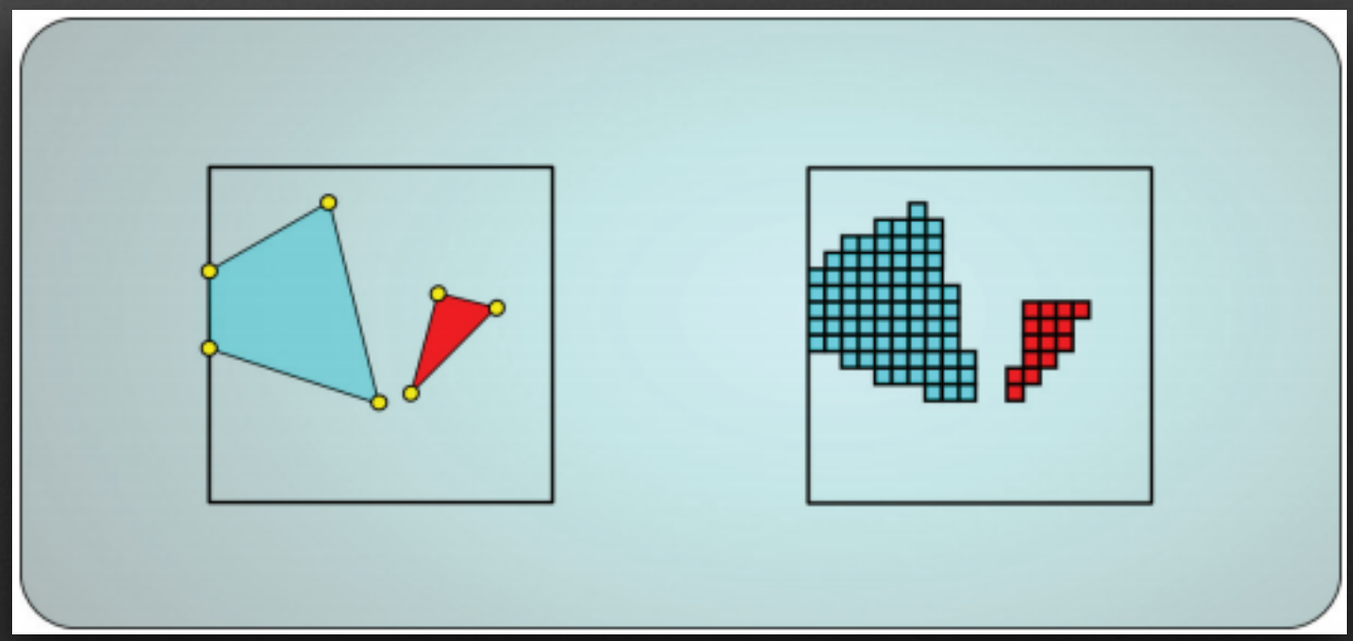
Viewport Transform

- map primitives from *NDC* to pixel coordinates;



Rasterization

- fragment generation
    - determine which pixels of the render target are covered by the current primitive;
        - the primitives must pass the *scissor test*
- attribute interpolation

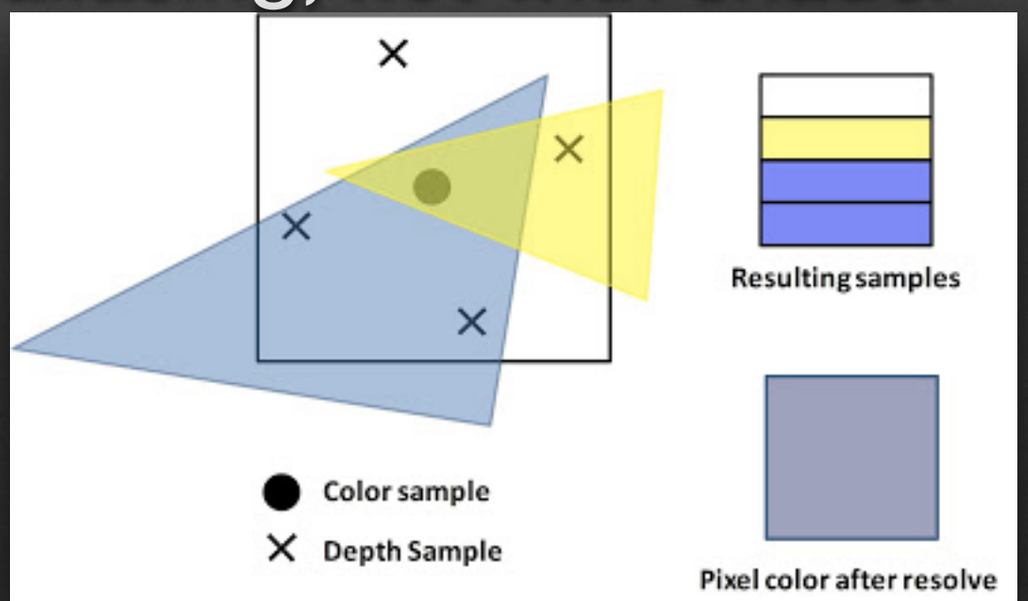


Rasterization

- One of the fundamental drawbacks of rasterization is aliasing artifacts;
    - *edge aliasing*
    - *shader aliasing*
- *super sampling anti-aliasing (SSAA)*
    - increase both rasterization and pixel shading;
    - expensive.

MSAA (multisampling anti-aliasing)

- the resolution used for rasterization and depth/stencil test is increased by an integral factor, but the pixel shader is still executed at the normal resolution;
- memory footprint of depth/stencil buffer & render target will increase by the MSAA factor;
- effectively deal with edge aliasing, not with shader aliasing



Rasterization Output

- fragment generation
- fragment data
    - each of the attributes that are passed into the rasterizer are interpolated

Pixel Shader

- fragments are passed to the Pixel Shader after generated by the rasterizer;

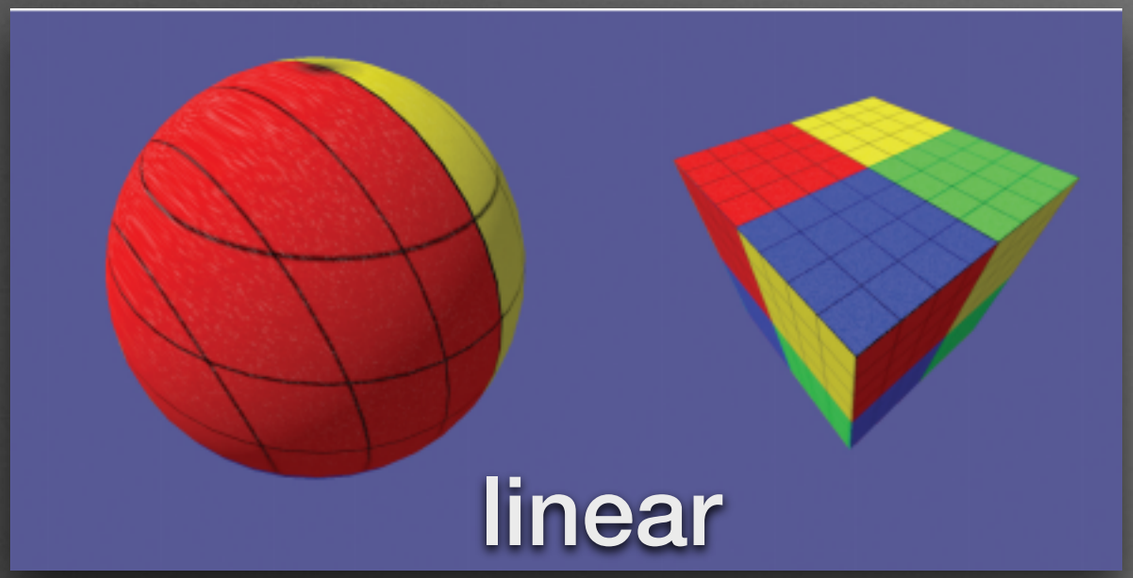
- PS processes each fragment individually;

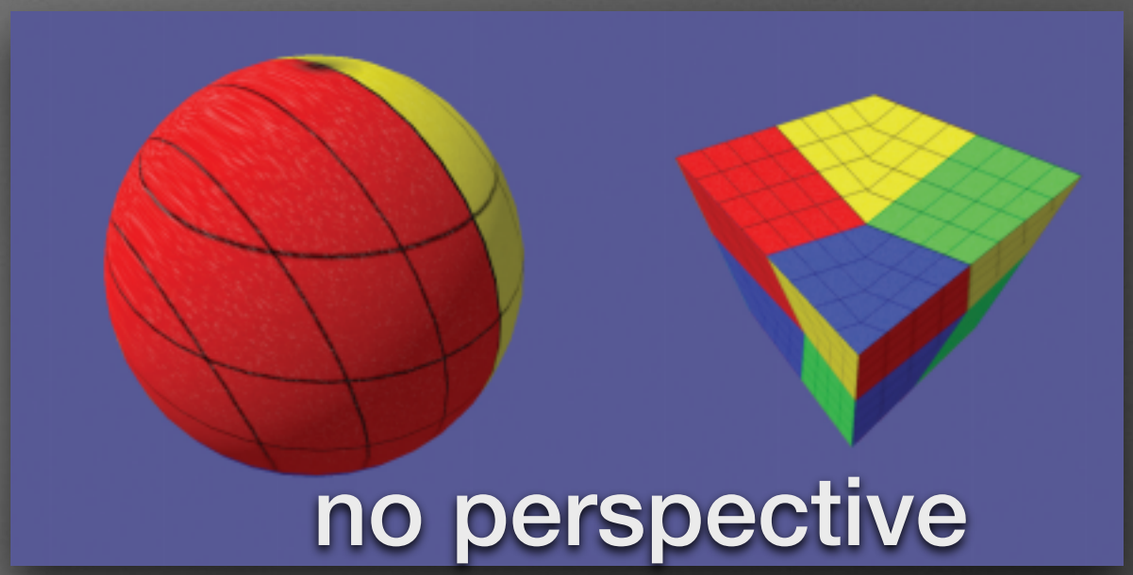
PS Input

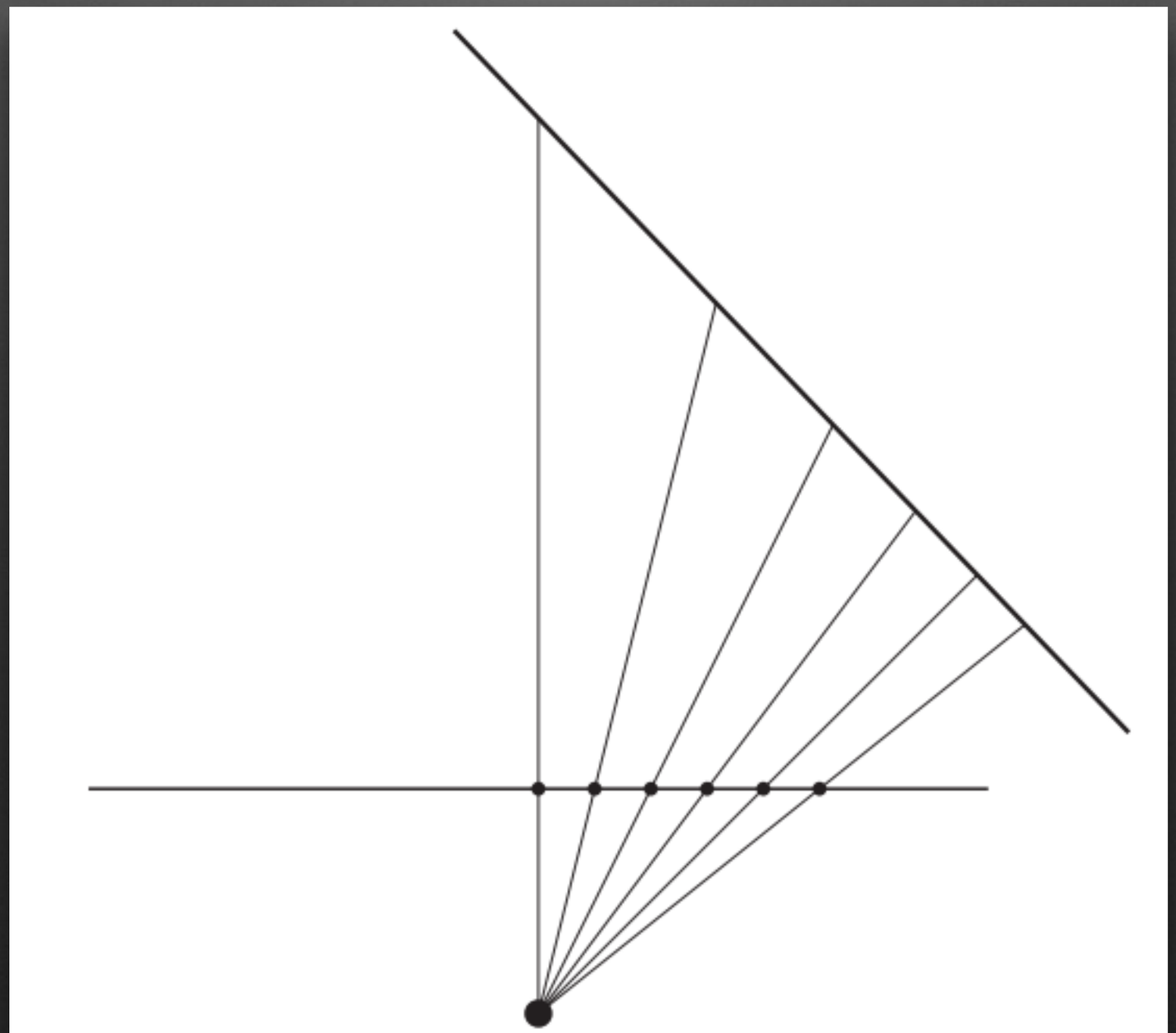
- Attribute Interpolation
  - three pieces of information is required for interpolating:
    - the attributes of vertices
    - the position of the point requires interpolation
    - interpolation technique

Interpolation techniques

MODE	DESCRIPTION
linear	Provides linear, perspective-correct interpolation. The interpolation is based on the center of the pixel
centroid	Provides linear, perspective-correct interpolation. The interpolation is based on the centroid of the covered are of the pixel
no perspective	Provides linear interpolation without accounting for perspective effects. The interpolation is based on the center of pixel


linear


no perspective



PS Input

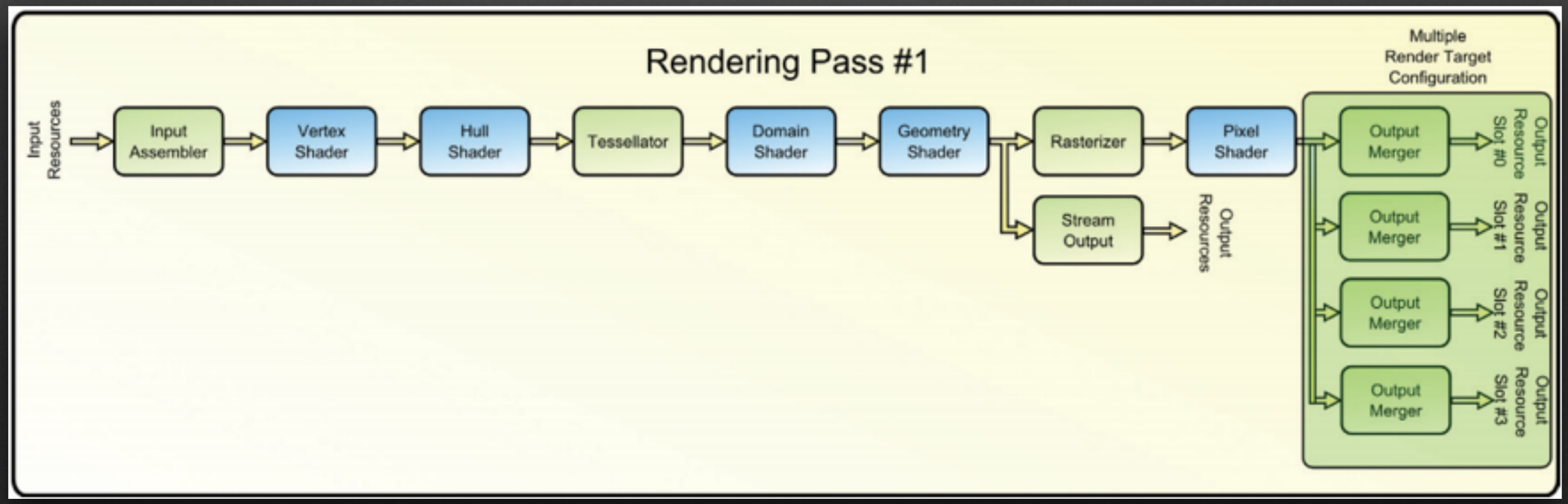
- Fragment Location
- Geometric Orientation

PS Configuration

- Common shader core resource configuration;
- Early Depth Stencil Test

PS Processing

- Multiple render targets (*MRTs*)
    - a single pixel shader invocation calculates and outputs a color to write to multiple render targets
    - save the cost of processing the geometric data multiple times



PS Processing

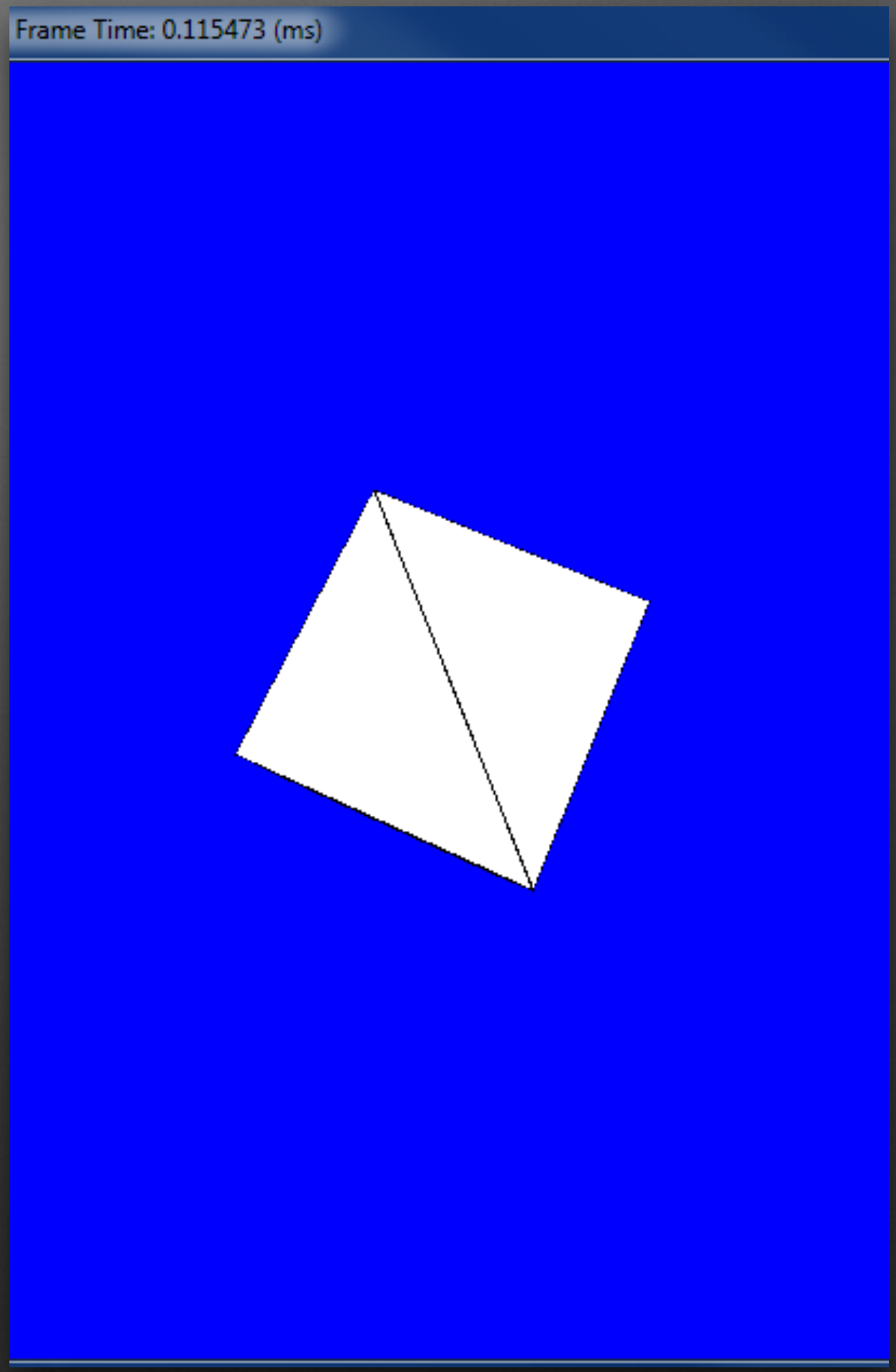
- Modifying depth values
  - the pixel shader can output a new depth value;
  - otherwise, the depth value generated in the rasterizer is passed to the output merger.

PS Output

- Because Output merger can only process color & depth value, PS output is limited to color and depth;

Demo

Drawing a wireframe on top of a shaded mesh



Output Merger

- final stop in the pipeline;
- receive color and depth results from the pixel shader, then merges results into the render targets bound to it for output;
- depth and stencil testing;

OM Input

- the primary input is the *color* values that were produced by the pixel shader;

- the second major input is the fragment's *depth* value, either from rasterizer or pixel shader.

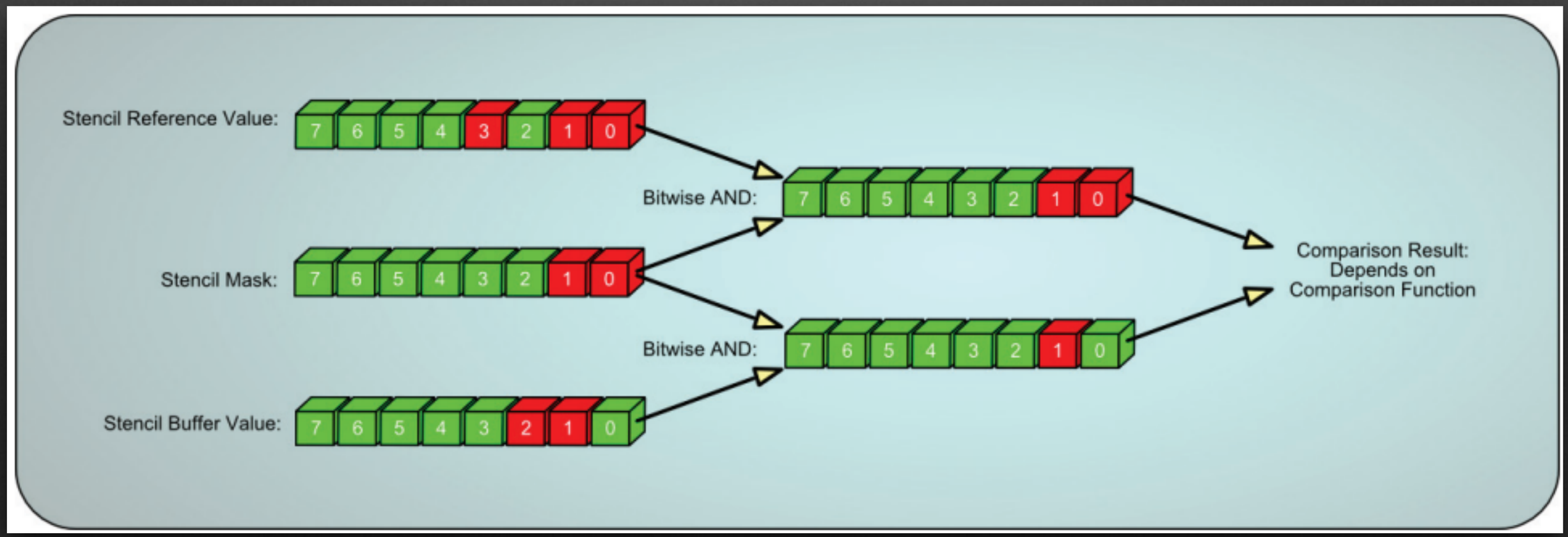
OM Configuration

- Depth Stencil State
- Blend State

OM Processing

- Visibility tests
  - stencil test

(StencilRef & StencilMask) *CompFunc* (StencilBufferValue & StencilMask)



OM Processing

- Visibility tests
    - depth test
        - if depth test fails, fragment will be discarded;
        - if both depth & stencil test passes, the depth value can be updated back to depth buffer, depends on pipeline settings;

OM Processing

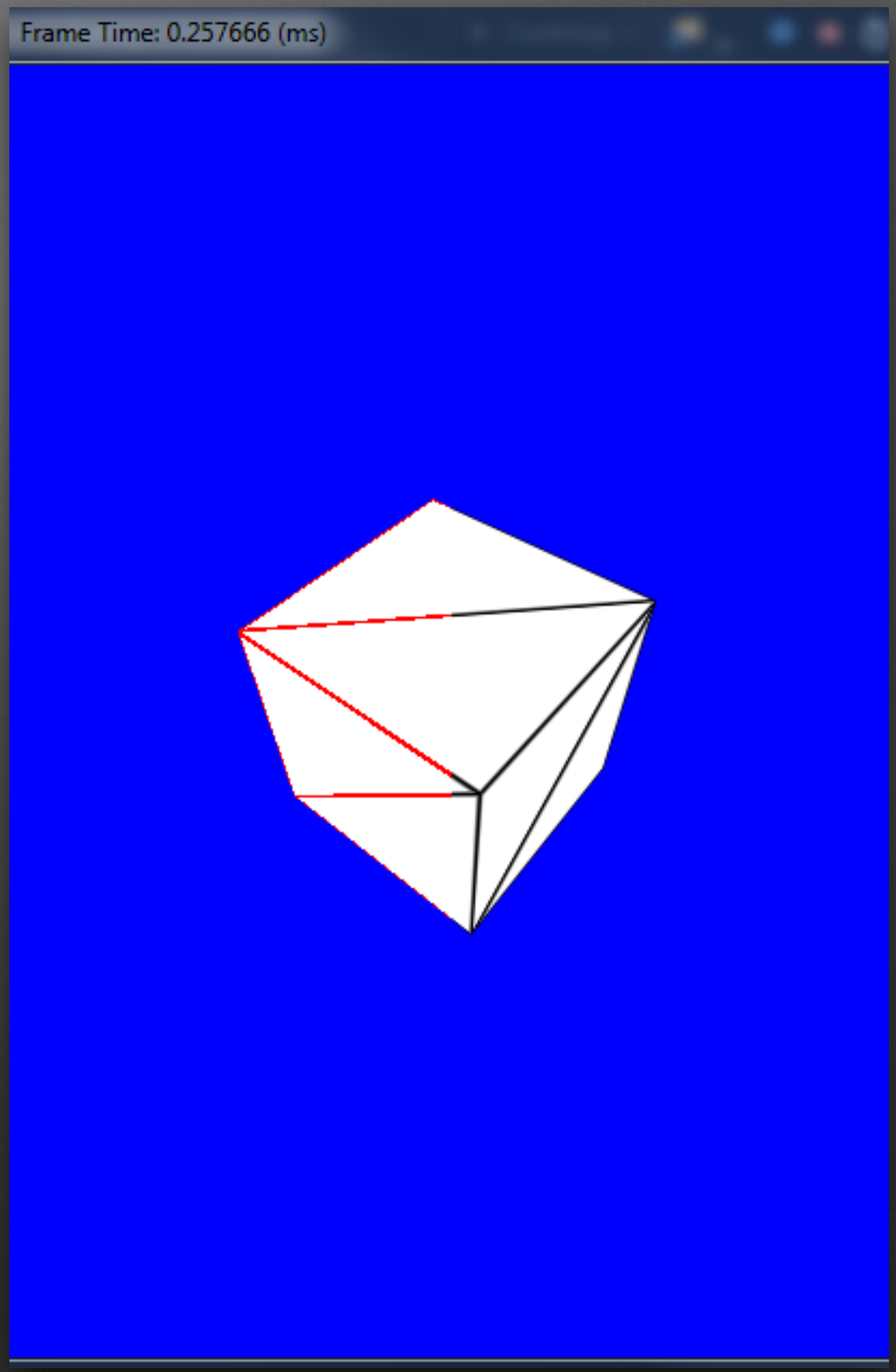
- Blending operation
    - traditionally used to perform alpha-blending;

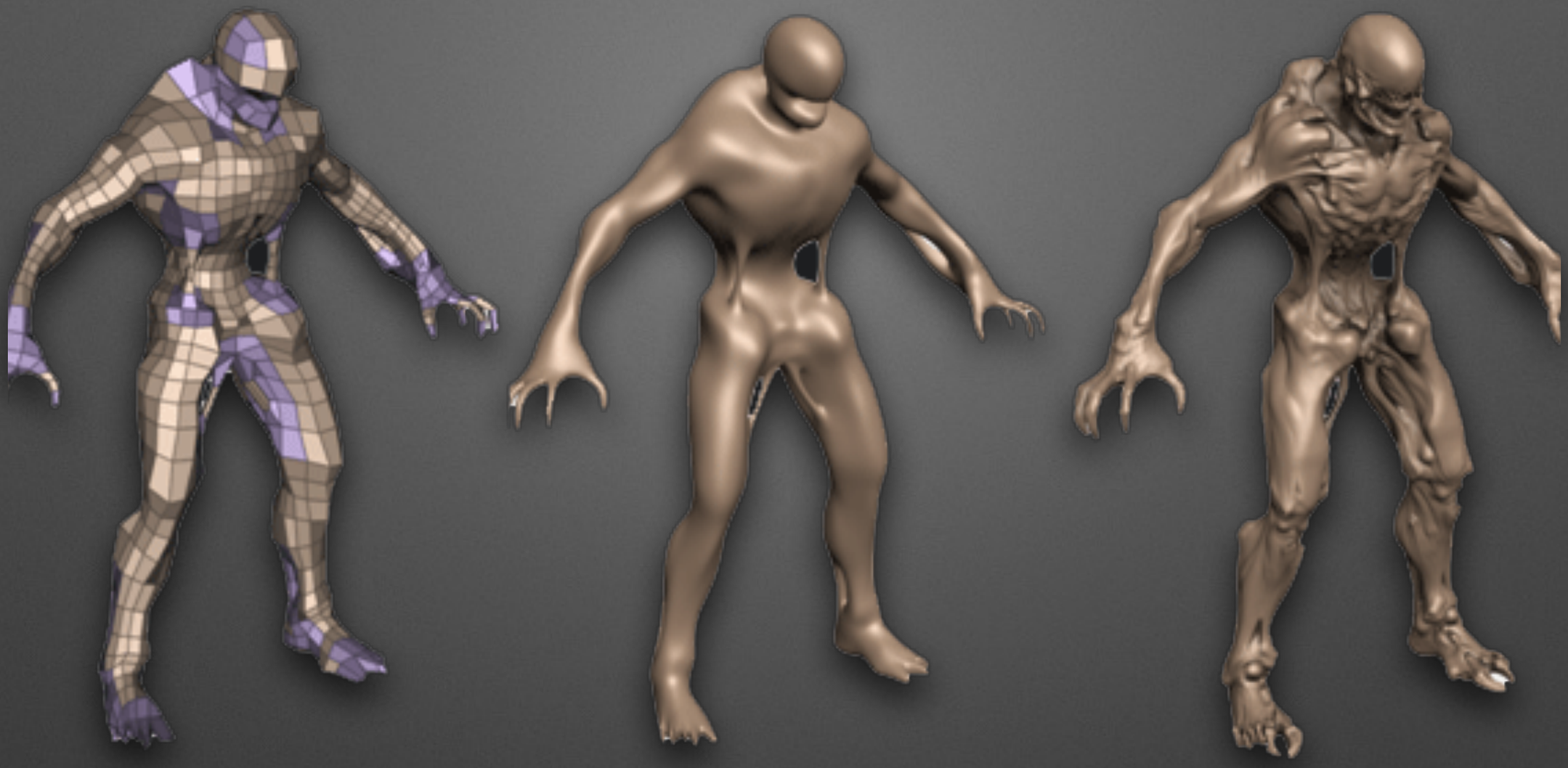
OM Output

- Render target
- Depth Stencil target
- When MSAA is in-use, subsamples have to be combined into single values

Demo

Anti-Aliasing





Tessellation Pipeline

Tessellation Pipeline Overview

- Tessellation: using many smaller pieces of geometry to create, without gaps or overlaps, a larger complete surface;

Impact on pipeline

- Input Assembler
    - functions exactly the same;
    - needs *vertex declaration , vertex buffer, index buffer;*
    - the topology will be changed to *control points;*
    - each primitive will contain many control points: [1, 32].

- Vertex Shader
  - VS no longer has to output clip-space vertex;
  - common use-case for VS now is animation;

- Hull Shader
    - made up of two functions, *hull shader* & *constant function*;
    - constant function: executed once per patch;
    - hull shader: executed once per *output* control point;
    - both have full visibility of all control points output by VS.

- Tessellator
    - fixed function, operates as a black-box;
    - two tessellation factor input from the *constant function* in HS;
    - output a set of weights corresponding to the primitive topology declared in the hull shader: line, triangle, quad.
    - handles the necessary winding and relations of the new created vertices, so they can correctly rendered.

- Domain Shader
    - can see all the control points;
    - its job is to take the point on the line/triangle/quad domain provided by the tessellator and use the control points provided by the hull shader to create a complete new, renderable vertex.

- Geometry Shader
    - unchanged;
    - has no knowledge of the tessellation work;
        - its execution number is related to the tessellation factor.
- Rasterization and anything after
    - unchanged.

Parameters for Tessellation

- As shader author, you have two way to control the increase of the tessellation details:
  - alter the number of control points in HS;
  - change the tessellation factor output by constant function in HS.
- Balance the performance and quality

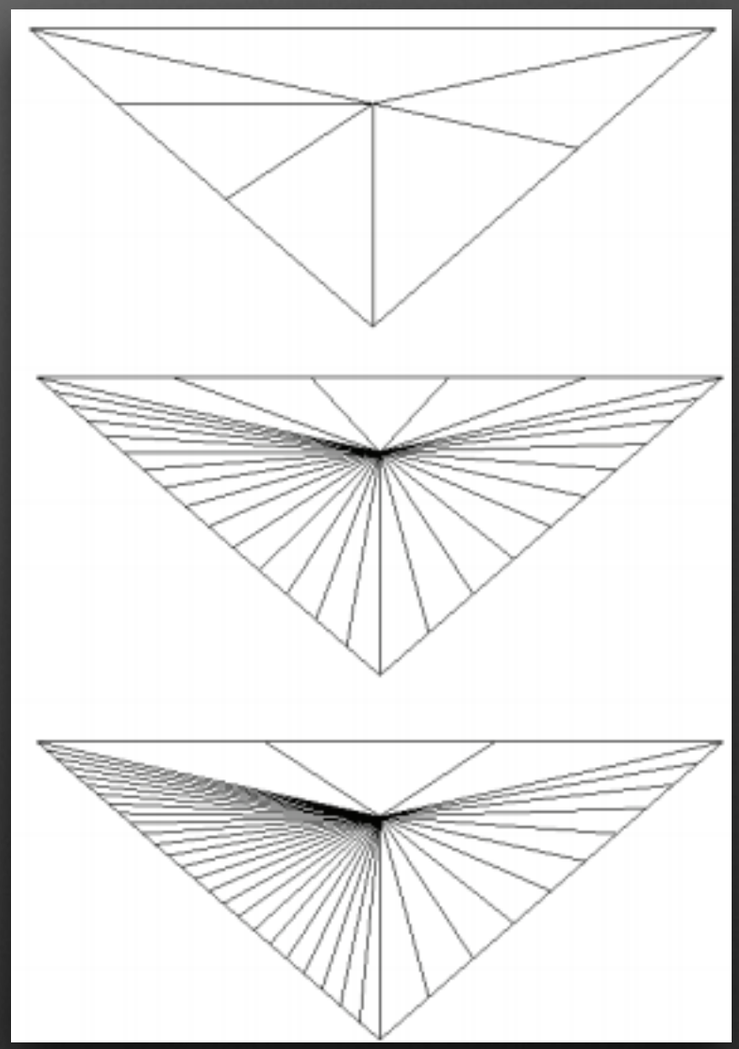
Tessellation factors

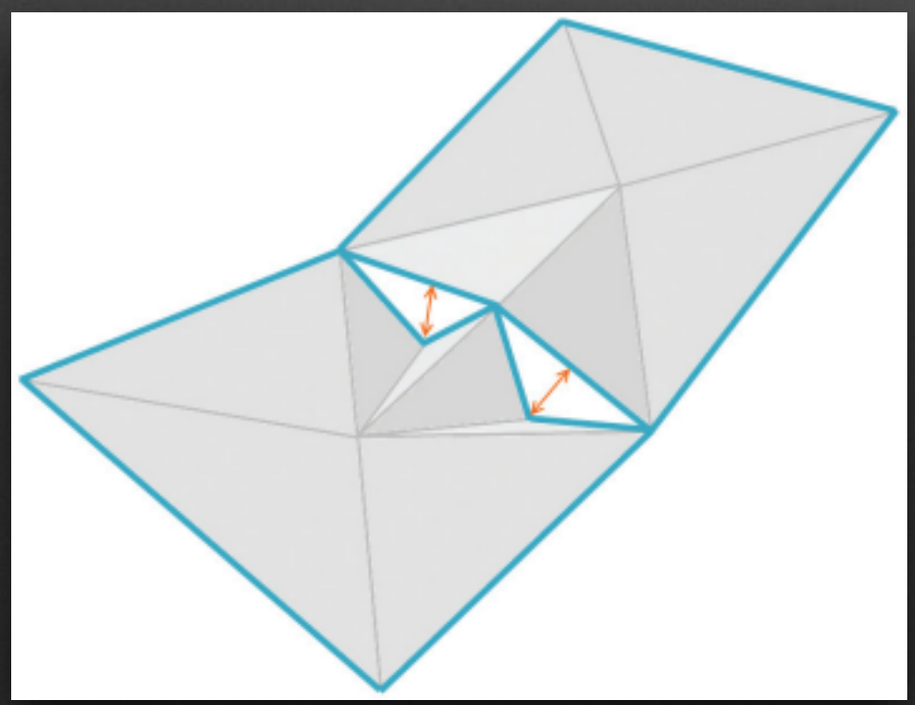
- Constant function in HS *MUST* output *edge factor* and *inside factor ,* which specify how finely to tessellate the current primitive;

	EDGE FACTOR	INSIDE FACTOR
LINE	2	0
TRIANGLE	3	1
QUAD	4	2

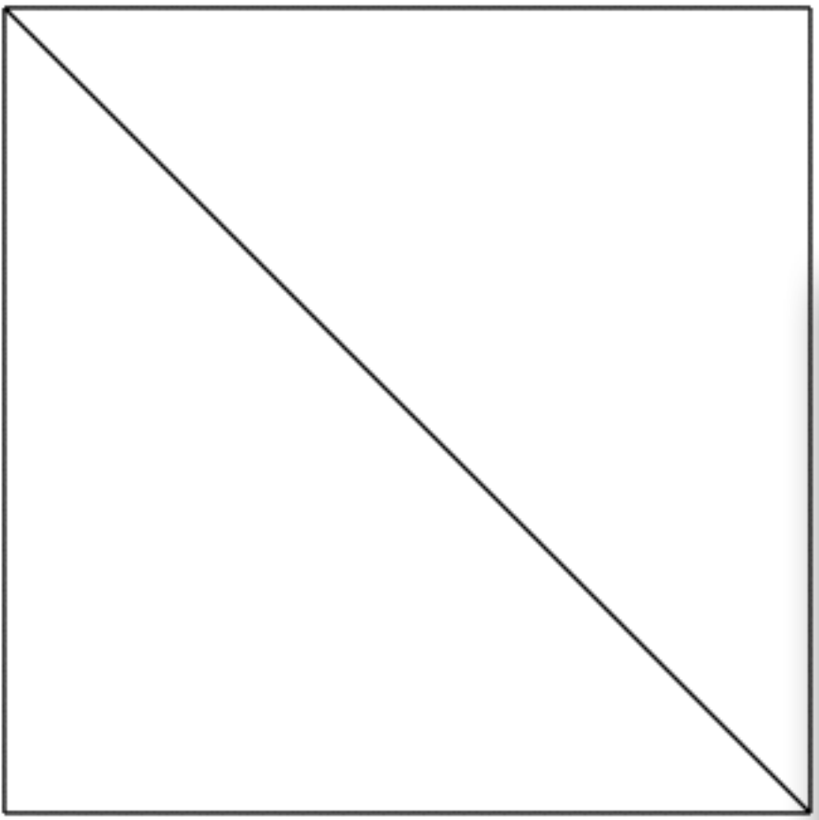
Effects of Tessellation factor

- Edge Factor
  - tessellator divides up each edge according to the matching edge factor;
  - mis-matching edge factor will generate gaps.

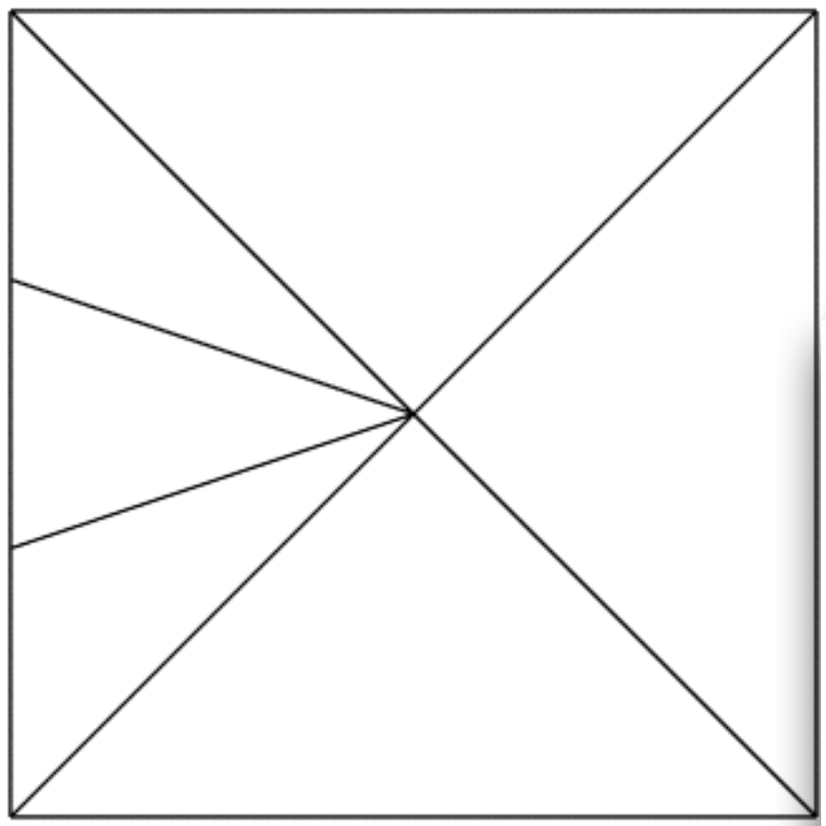




edge factor example

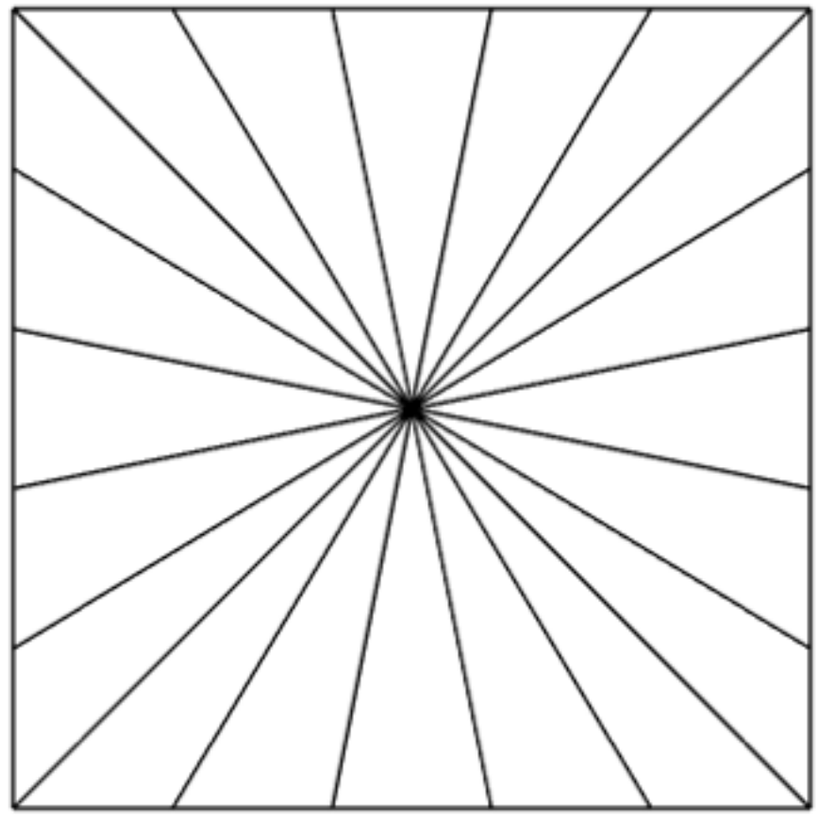


(1, 1, 1, 1)

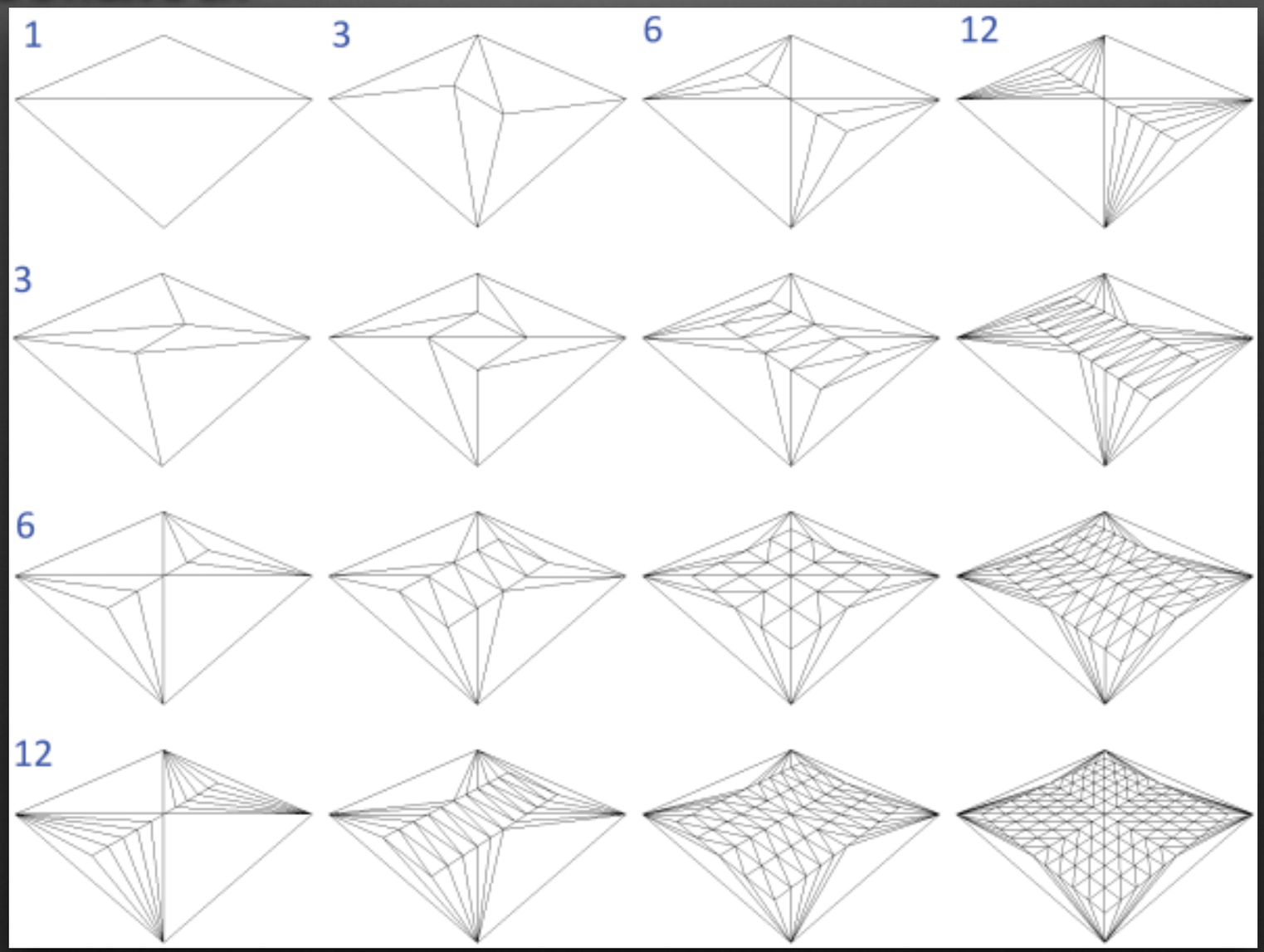


(3, 1, 1, 1)

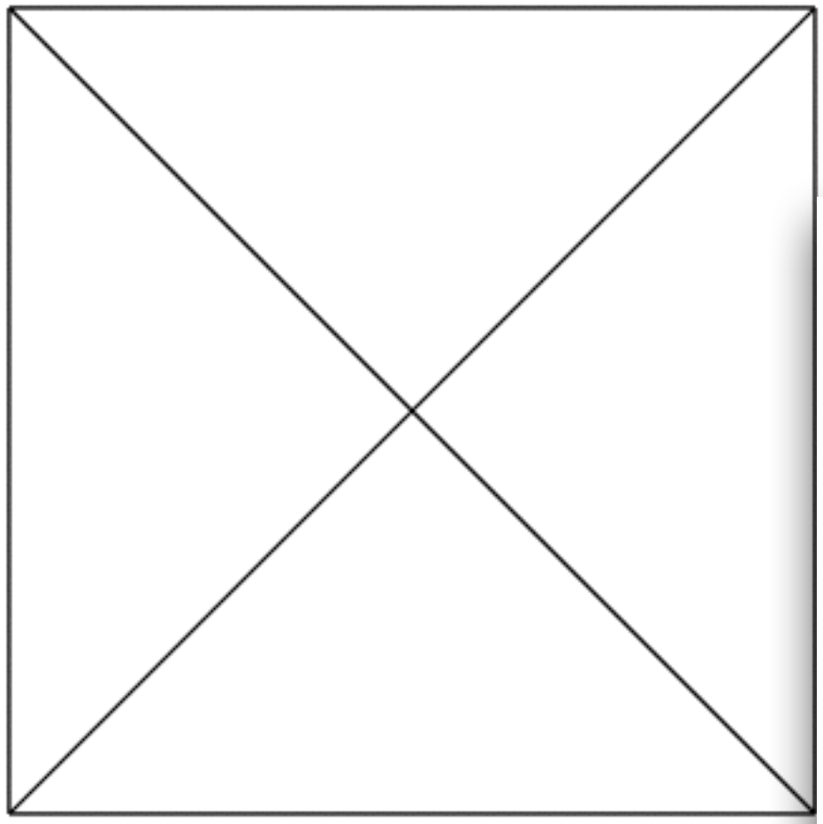
(5, 5, 5, 5)



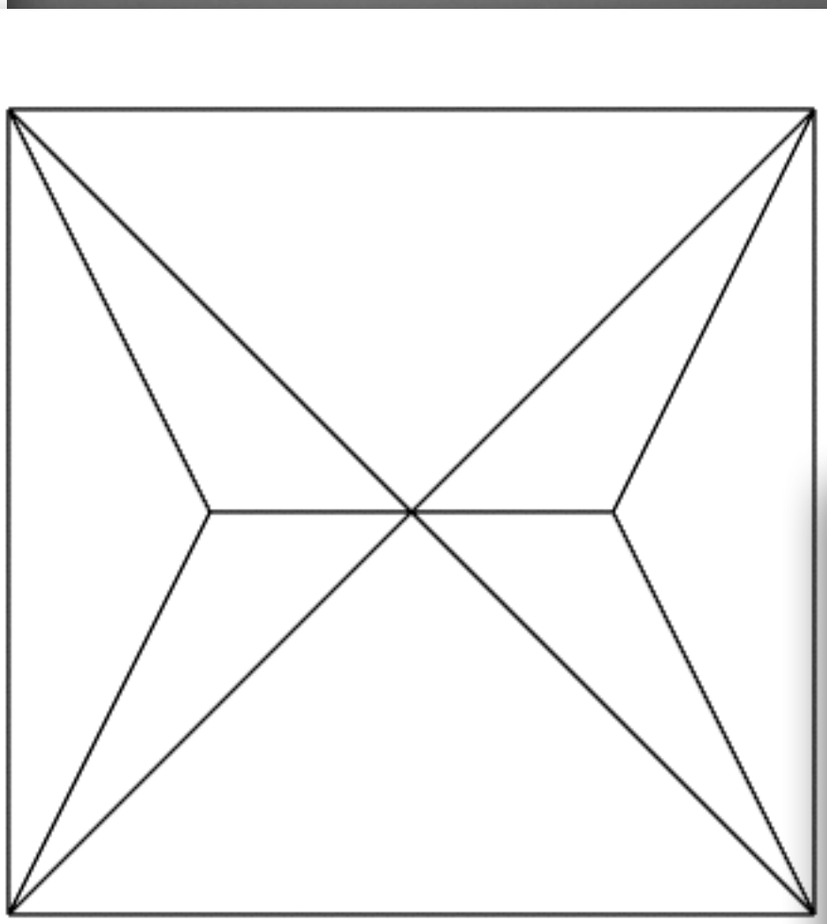
- Inside Factor
  - increase the internal tessellation factors;
  - makes the inner area of the surface more densely tessellated.



Inside factor example

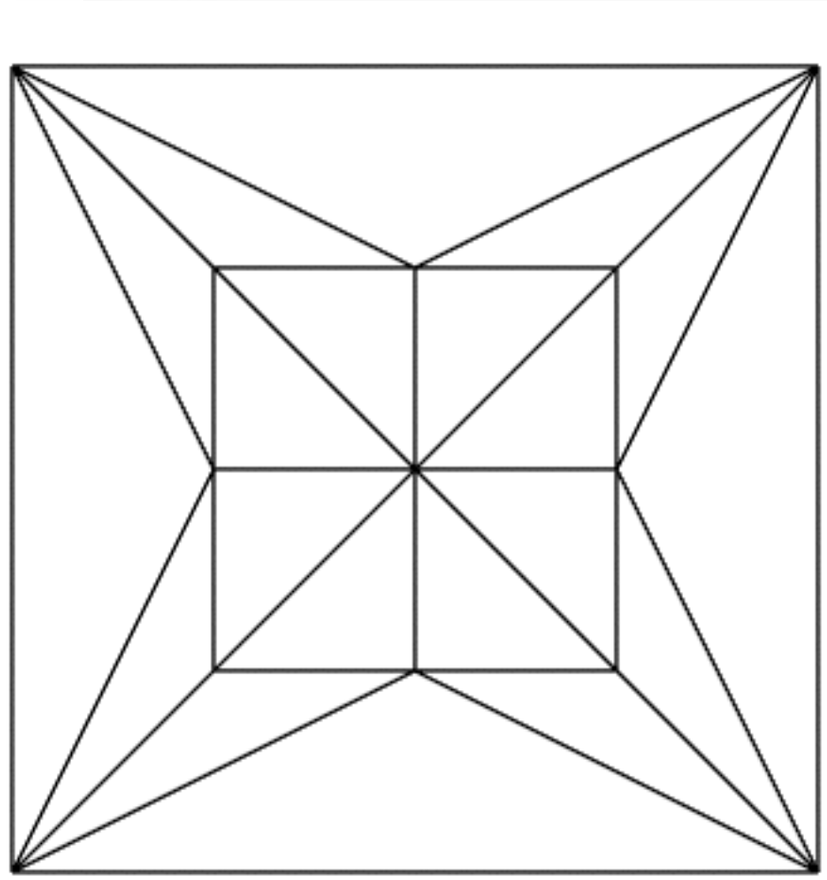


(2, 1)



(4, 1)

(4, 4)



- Partitioning
    - declared as an attribute of the main hull shader, affect how tessellator interprets the input tessellation factor from constant function;
    - can be *integer, fractional_even, fractional_odd, pow2;*
        - integer: rounds to the nearest integer
        - fractional_even: rounds to nearest even number;
        - fractional_odd: rounds to nearest odd number;
        - pow2: rounds to the number of the power of 2.

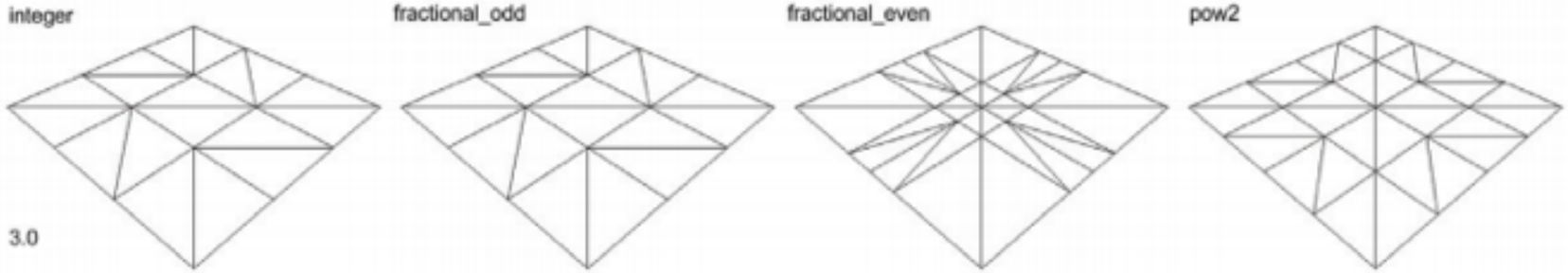
integer

fractional_odd

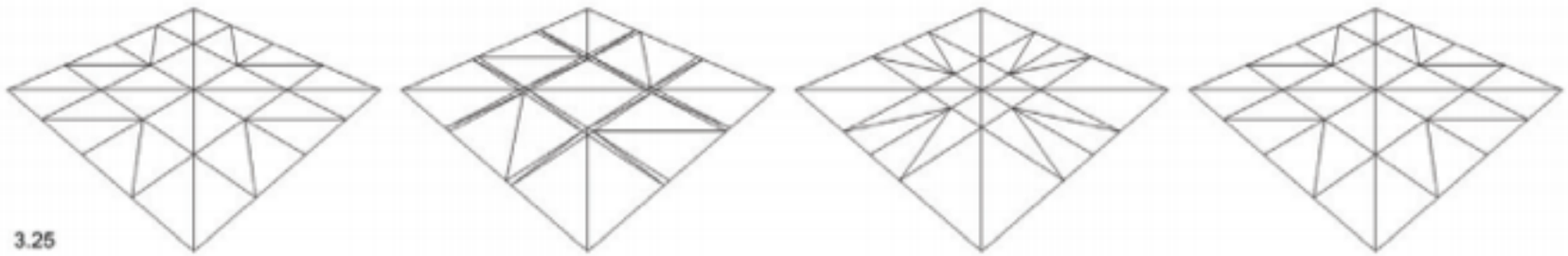
fractional_even

pow2

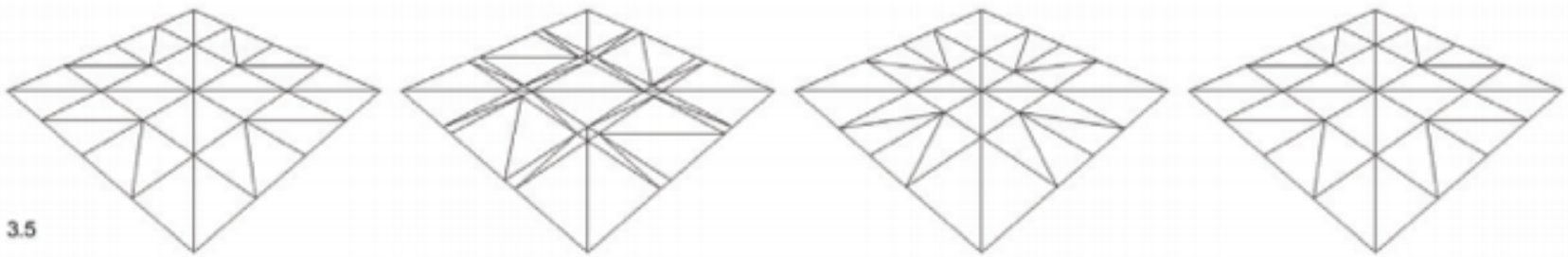
3.0



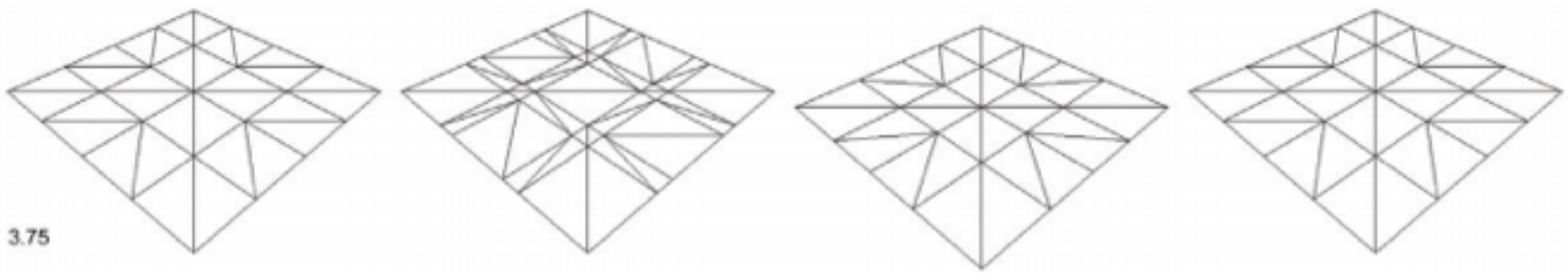
3.25



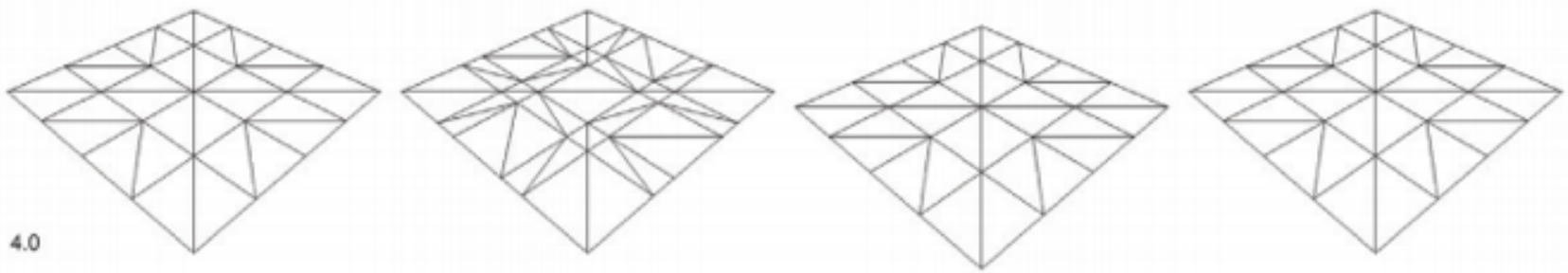
3.5



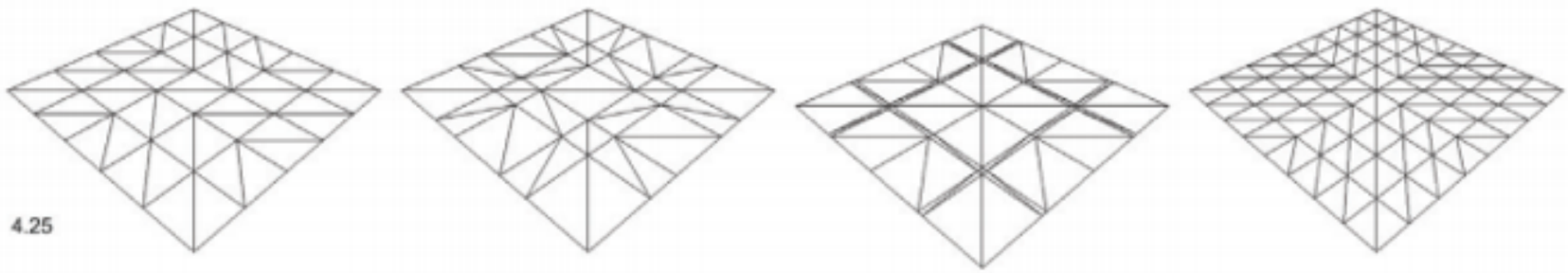
3.75



4.0



4.25



Hull Shader

- Hull shader program: invoked once for each *output control point*.
  - must create one control point for each invocation;
  - has access to all of the input control points.
- Constant function: only executed once for each complete patch.

HS Input

- a set of processed control points produced by the VS;
- primitive stream information produced by IA;
- some system semantics
    - hull shader: SV_OutputControlPointID, SV_PrimitiveID;
    - constant function: SV_PrimitiveID;

HS configuration

- Common shader core resource configuration;
- Function Attributes
    - *domain*: specify the domain of the primitive being tessellated;
    - *partitioning*: specify the method to tessellate with;
    - *output topology*: specify the type of primitives to be created from the tessellation;
    - *output control points*: specify the number of points created by the hull shader program;
    - *patchconstantfunc*: specify the name of the constant function;
    - *maxtessfactor*: specify the largest tessellation factor this shader can produce.

HS processing

- Hull Shader Program
    - Tessellation algorithm setup
    - Tessellation algorithm level of detail
        - Generic control point calculation
- Tessellation Factor Calculation
    - Fine level of detail
    - Generic control patch calculations

HS output

- output control point of the control patch
- tessellation factors

Tessellator

- convert the requested amount of tessellation into a group of coordinate points within the current “domain”
- domain available: line, triangle, quad.

Tessellator input

- tessellation factors produced by the hull shader constant function;

Tessellator configuration

- configured by the function attributes in the hull shader
    - Domain Attribute (Must)
    - Partitioning Attribute (Must)
    - Output Topology Attribute (Must)
    - Max Tessellation Factor Attribute

Tessellator processing

- Sample Locations
    - tessellation points are chosen based on the domain type and tessellation factors;
- Primitive Generation
    - when the tessellation stages are active, IA will specify the topology of control points, tessellator will specify the primitive information.

Tessellator Output

- output every newly generated domain points to domain shader through system semantic (SV_DomainLocation);
- primitive topology information will passed beyond domain shader.

Domain Shader

- final stop in the tessellation pipeline
- invoked once for each coordinate point produced by the tessellator;
- receive input from both hull shader and tessellator
- produce vertices

DS input

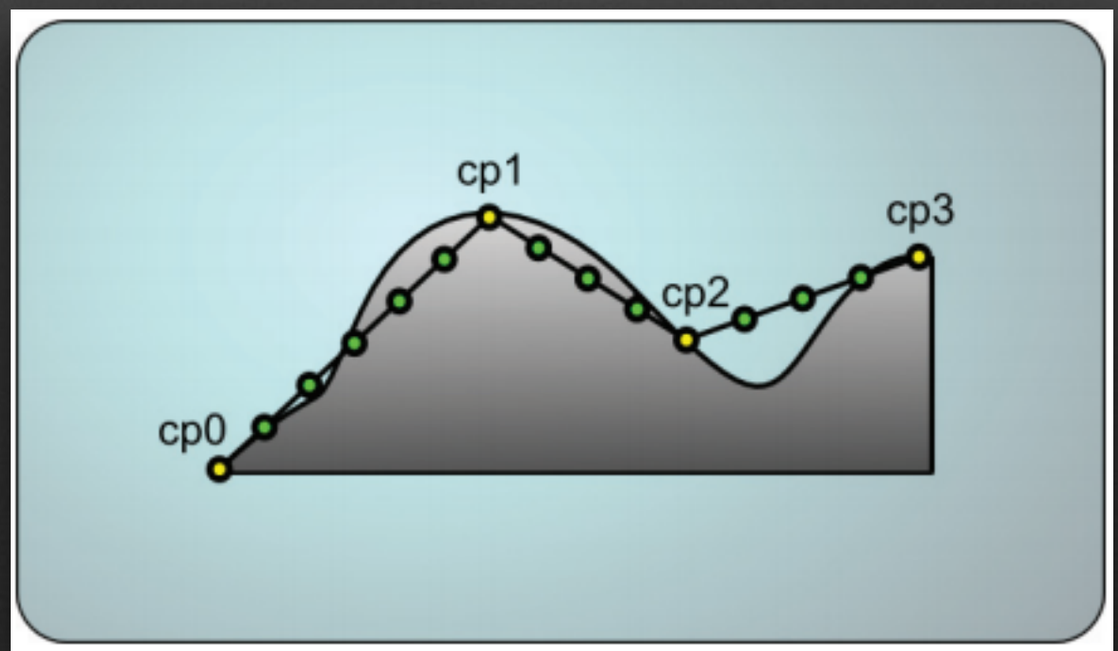
- complete control patch from HS;
- domain location from tessellator;
- constants produced by constant function from HS.

DS configuration

- Common shader core resource configuration;

DS processing

- the primary job of the DS is to create vertices using its input data;
- we can think of control patch as a set of parameters which define a virtual surface, each coordinate points produces by the tessellator as a selected location where we want to sample the virtual surface.



DS output

- the newly created vertices are returned by the domain shader program and passed to the next stage.