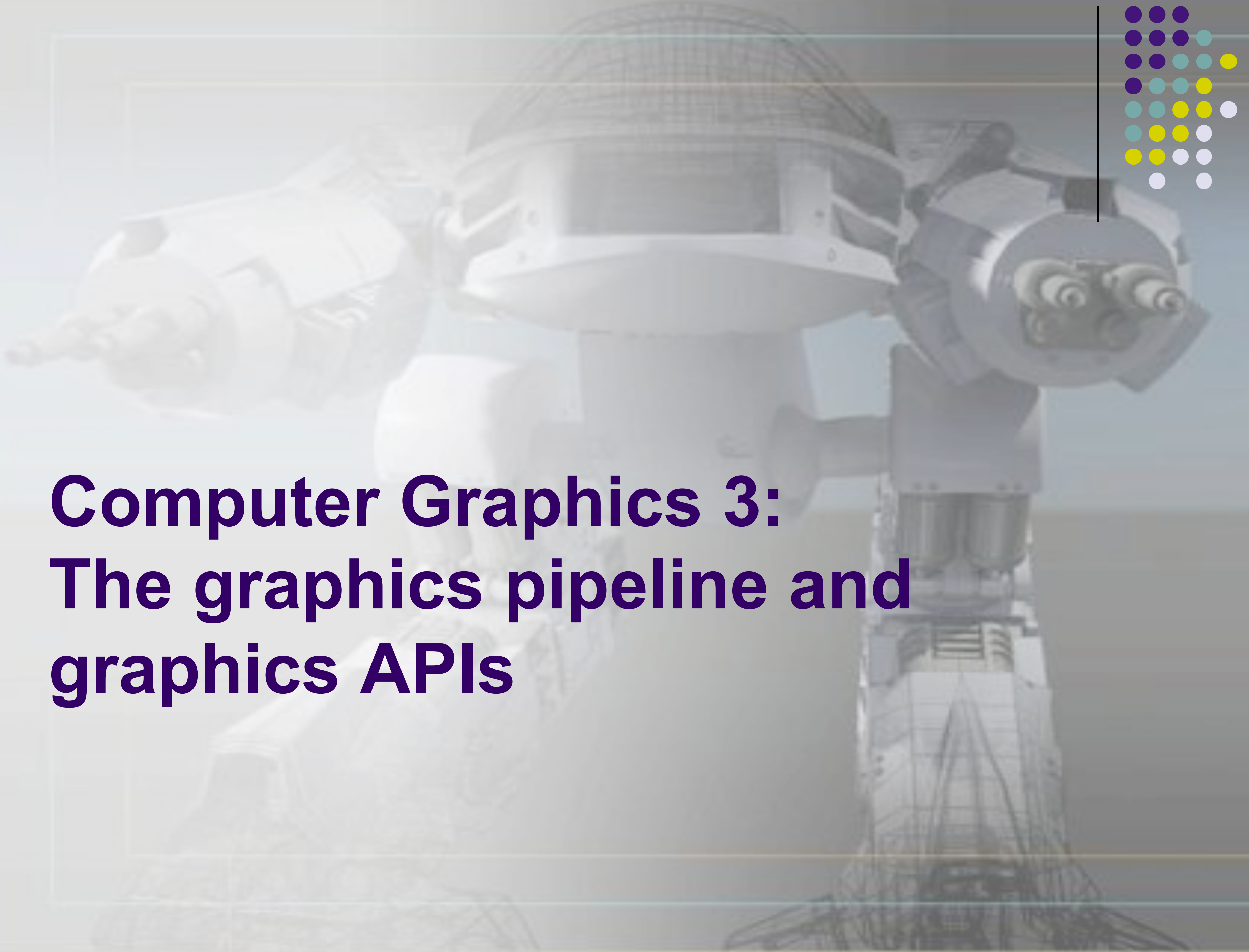
Computer Graphics 3: The graphics pipeline and graphics APIs



Programmer view of the graphics pipeline

- numerous ways that a user can interact with a graphics system
- For instance a painting program in the user sees menus and icons that represent possible actions
- We can also interact with the graphics pipeline through programming using an API

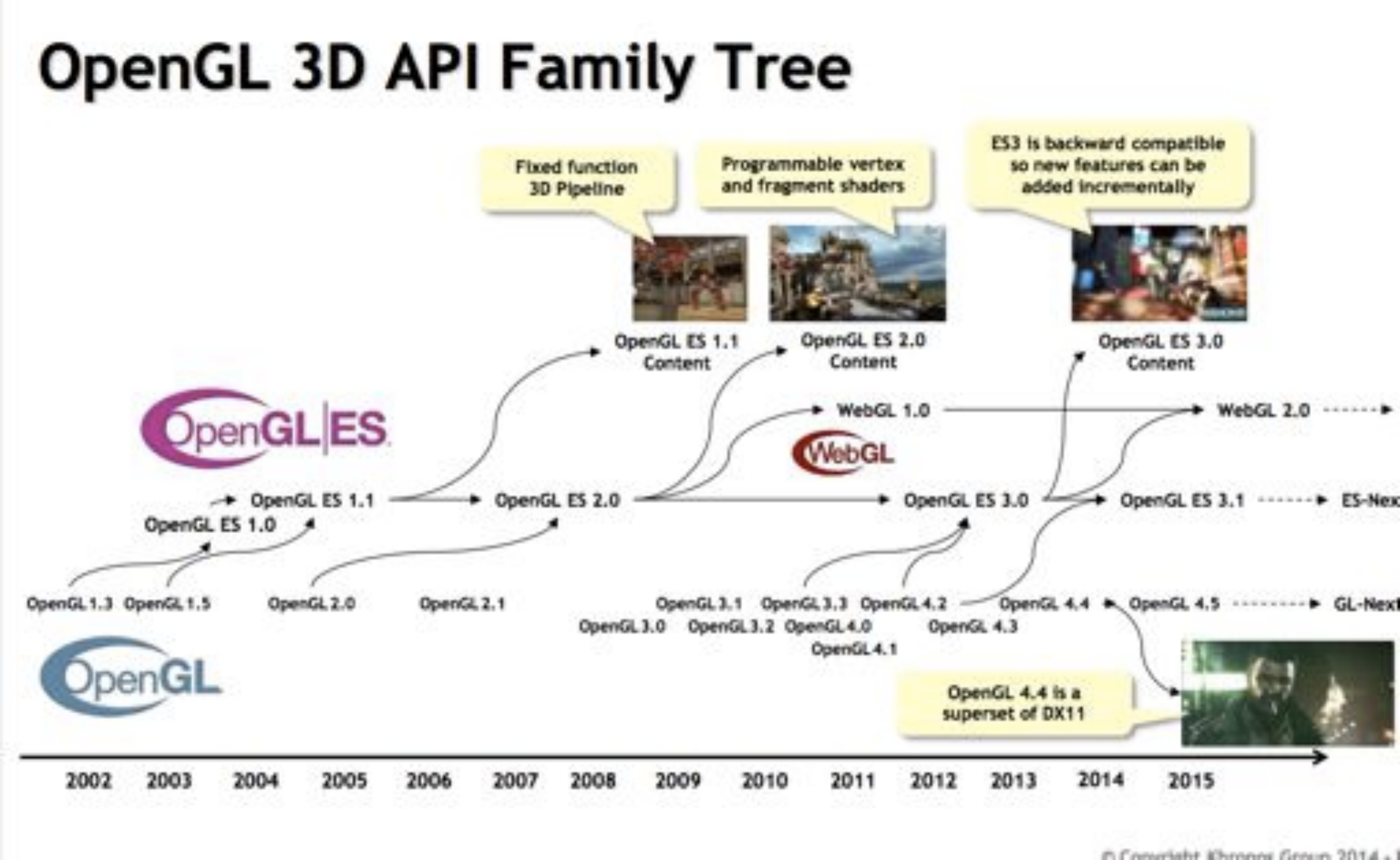
Graphics API

- There are many Graphics APIs
- The 2 most common are Open GL and DirectX and its 3d version Direct3D
- OpenGL is cross platform, open standard
- Originally Created by Silicon graphics, the responsibility for maintaining the standard has been turned over to the Khronos group

Graphics API

- There are many Graphics APIs
- The 2 most common are Open GL and DirectX and its 3d version Direct3D
- OpenGL is cross platform, open standard
- Originally Created by Silicon graphics, the responsibility for maintaining the standard has been turned over to the Khronos group

Graphics API



OpenGL 3D API Family Tree

Fixed function 3D Pipeline

Programmable vertex and fragment shaders

ES3 is backward compatible so new features can be added incrementally

OpenGL ES 1.1 Content

OpenGL ES 2.0 Content

OpenGL ES 3.0 Content

OpenGL|ES

WebGL 1.0 → WebGL 2.0 ------→

WebGL

OpenGL ES 1.0 → OpenGL ES 1.1 → OpenGL ES 2.0 → OpenGL ES 3.0 → OpenGL ES 3.1 ------→ ES-Nex

OpenGL 1.3 OpenGL 1.5 OpenGL 2.0 OpenGL 2.1 OpenGL 3.0 OpenGL 3.1 OpenGL 3.2 OpenGL 3.3 OpenGL 4.0 OpenGL 4.1 OpenGL 4.2 OpenGL 4.3 OpenGL 4.4 → OpenGL 4.5 --------→ GL-Next

OpenGL

OpenGL 4.4 is a superset of DX11

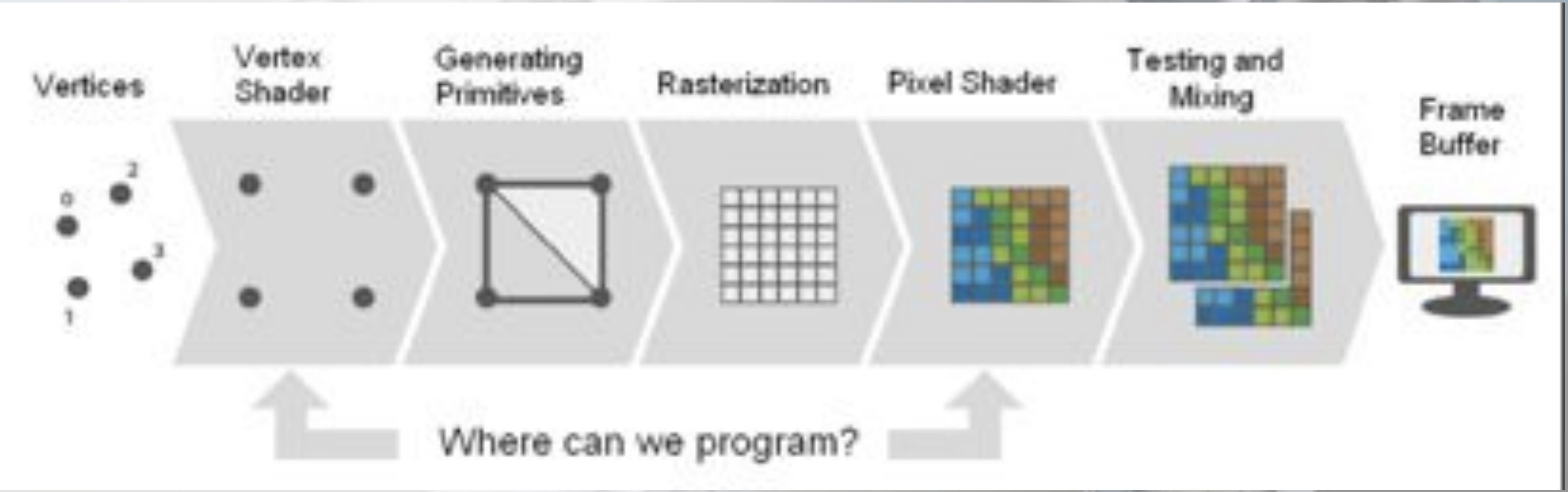
2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015

© Copyright Khronos Group 2014 -

Outline of rendering pipeline

- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Outline of graphics pipeline

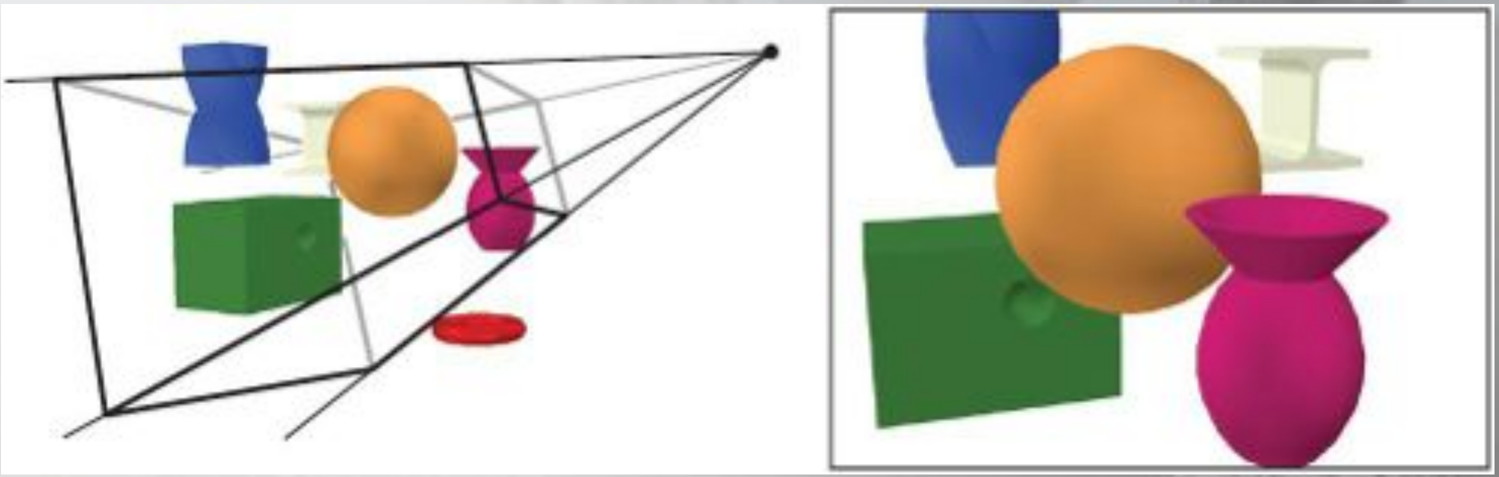


Rendering

- the process of generating an image given
  - a virtual camera
  - objects
  - light sources
- various techniques, e. g.
  - rasterization (topic of this course)
  - raytracing (topic of the course “Advanced Computer Graphics”)
- one of the major research topics in computer graphics
  - rendering
  - animation
  - geometry processing

Rasterization

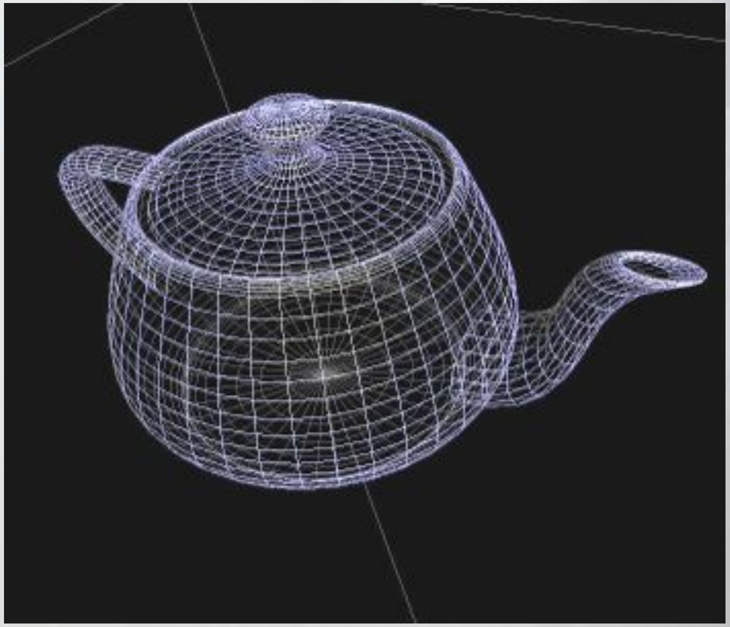
- rendering algorithm for generating 2D images from 3D scenes
- transforming geometric primitives such as lines and polygons into raster image representations, i. e. pixels

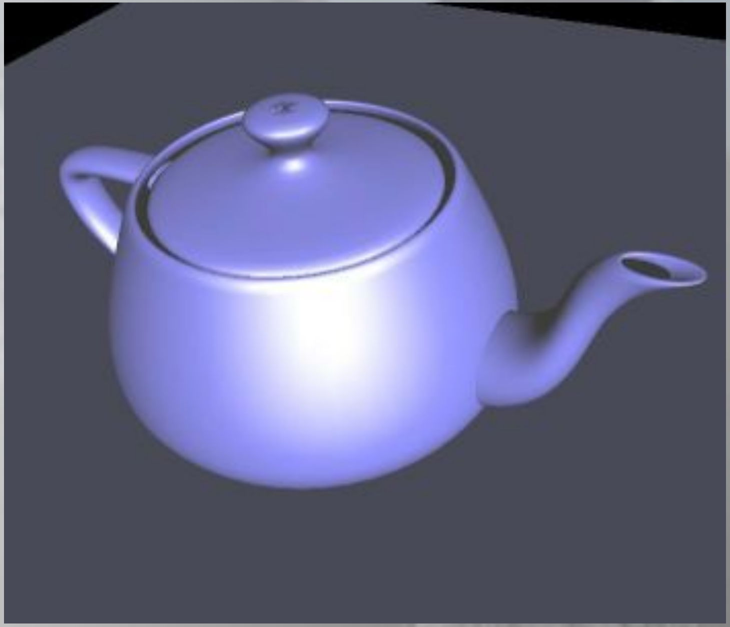


[Akenine-Moeller et al.: Real-time Rendering]

Rasterization

- 3D objects are approximately represented by vertices (points), lines, polygons
- these primitives are processed to obtain a 2D image





[Akenine-Moeller]

Rendering Pipeline

- processing stages comprise the rendering pipeline (graphics pipeline)
- supported by commodity graphics hardware
  - GPU - graphics processing unit
  - computes stages of the rasterization-based rendering pipeline
- OpenGL and DirectX are software interfaces to graphics hardware
  - this course focuses on concepts of the rendering pipeline
  - this course assumes OpenGL in implementation-specific details

Outline

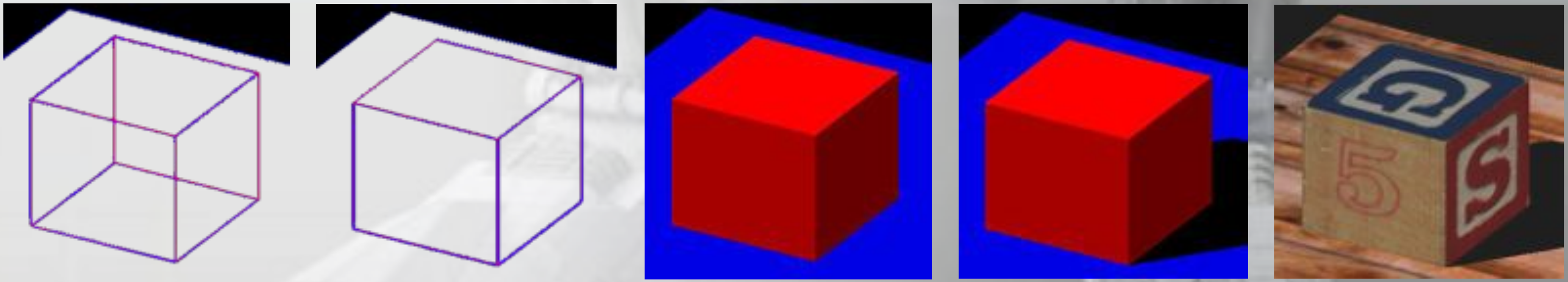
- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Rendering Pipeline - Task

- 3D input
  - a virtual camera
    - position, orientation, focal length
  - objects
    - points (vertex / vertices), lines, polygons
    - geometry and material properties (position, normal, color, texture coordinates)
  - light sources
    - direction, position, color, intensity
  - textures (images)
- 2D output
  - per-pixel color values in the framebuffer

Rendering Pipeline / Some Functionality

- resolving visibility
- evaluating a lighting model
- computing shadows (not core functionality)
- applying textures



visibility lighting model shadow texture

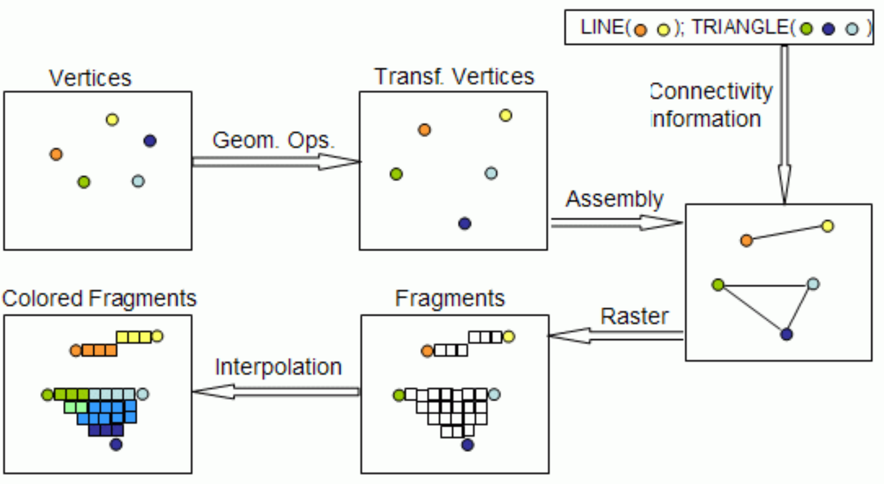
[Wright et al.: OpenGL SuperBible]

Rendering Pipeline Main Stages

- vertex processing / geometry stage / vertex shader
  - processes all vertices independently in the same way
  - performs transformations per vertex, computes lighting per vertex
- geometry shader
  - generates, modifies, discards primitives
- primitive assembly and rasterization / rasterization stage
  - assembles primitives such as points, lines, triangles
  - converts primitives into a raster image
  - generates fragments / pixel candidates
  - fragment attributes are interpolated from vertices of a primitive
- fragment processing / fragment shader
  - processes all fragments independently in the same way
  - fragments are processed, discarded or stored in the framebuffer

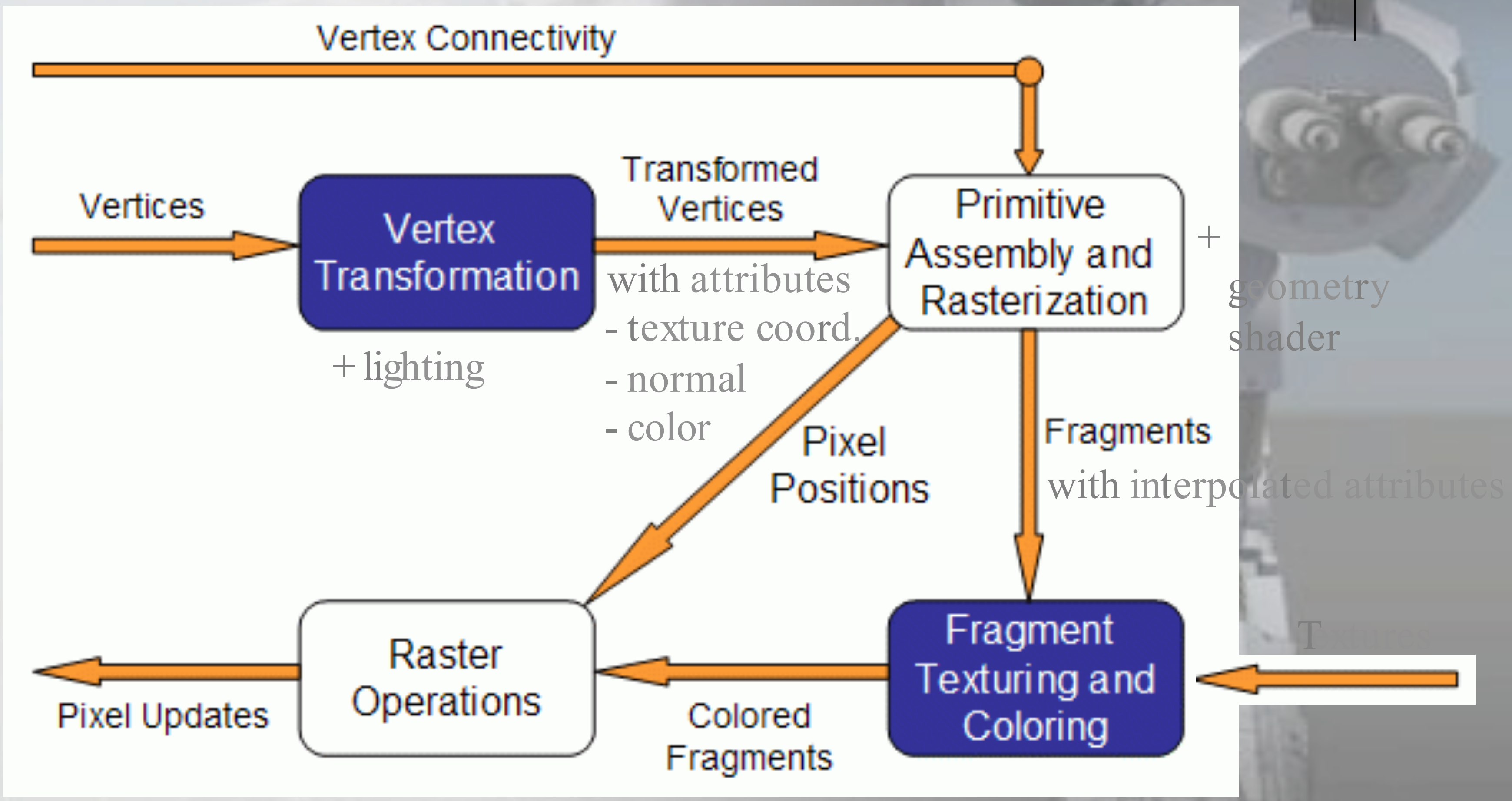
Rendering Pipeline Main Stages

- vertex position transform
- lighting per vertex
- primitive assembly, combine vertices to lines, polygons
- rasterization, computes pixel positions affected by a primitive
- fragment generation with interpolated attributes, e. g. color
- fragment processing (not illustrated), fragment is discarded or used to update the pixel information in the framebuffer, more than one fragment can be processed per pixel position



[Lighthouse 3D]

Rendering Pipeline Main Stages



Vertex Connectivity

Vertices

Vertex Transformation

+ lighting

Transformed Vertices

with attributes
- texture coord.
- normal
- color

Primitive Assembly and Rasterization

+ geometry shader

Pixel Positions

Fragments

with interpolated attributes

Fragment Texturing and Coloring

Textures

Colored Fragments

Raster Operations

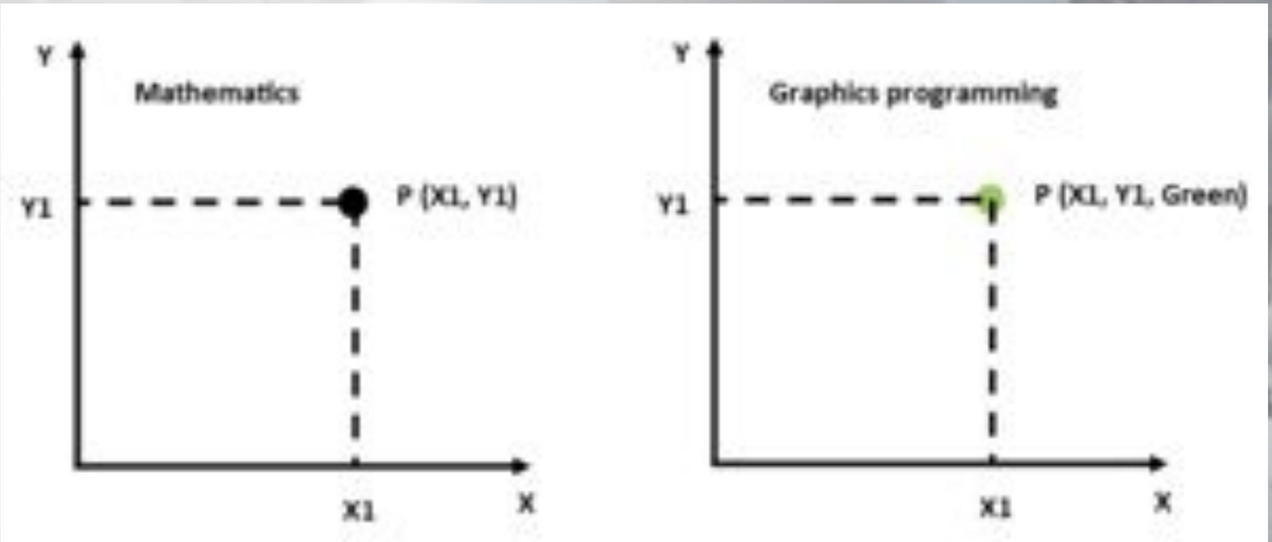
Pixel Updates

[Lighthouse 3D]

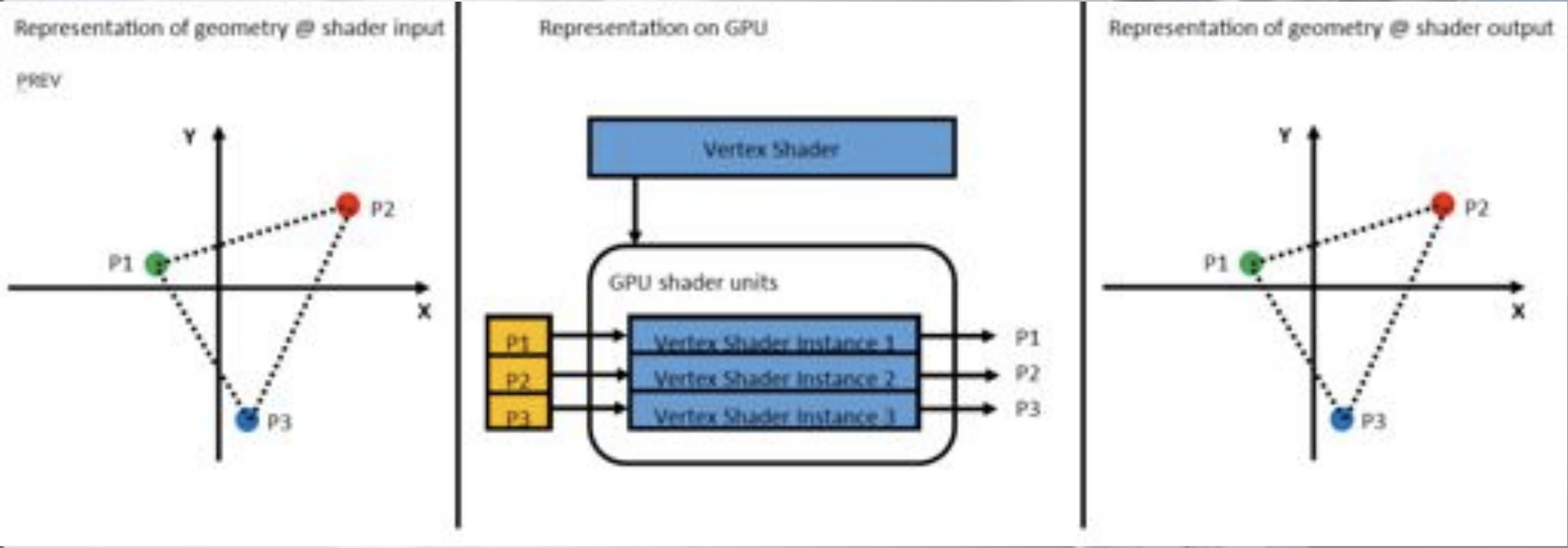
Outline

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

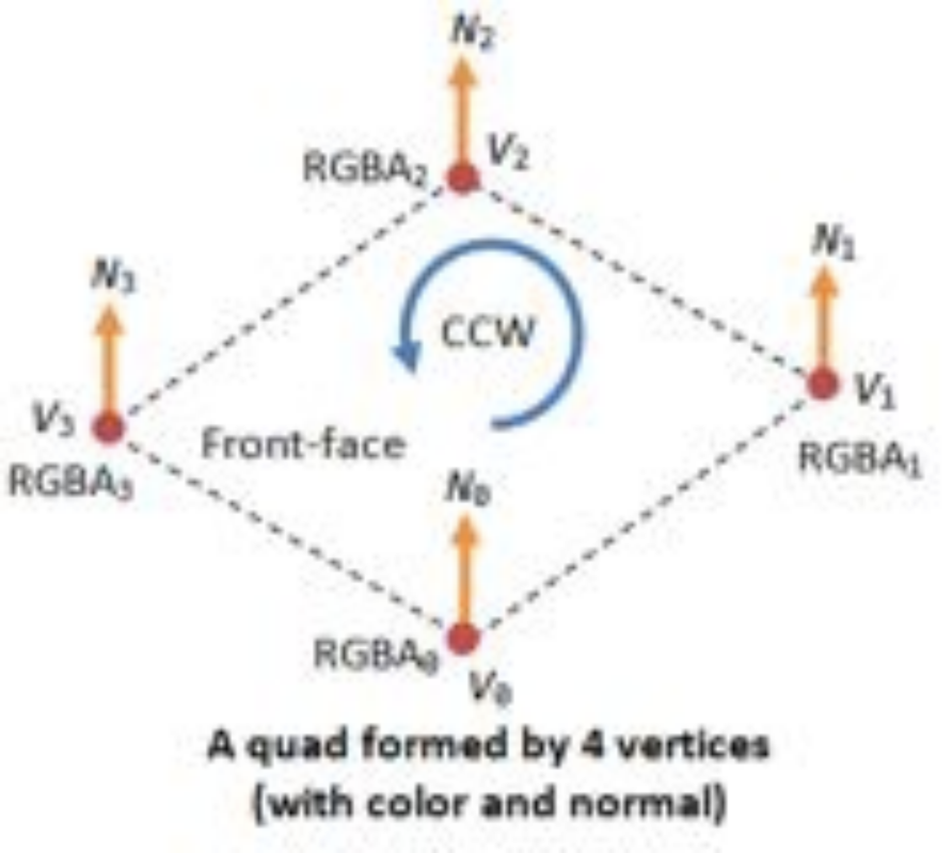
Vertex


Y
Mathematics
Y1
P (X1, Y1)
X1
X
Y
Graphics programming
Y1
P (X1, Y1, Green)
X1
X

Shaders - Vertex Shaders



Primitive Assembly



A quad formed by 4 vertices
(with color and normal)

Rasterization

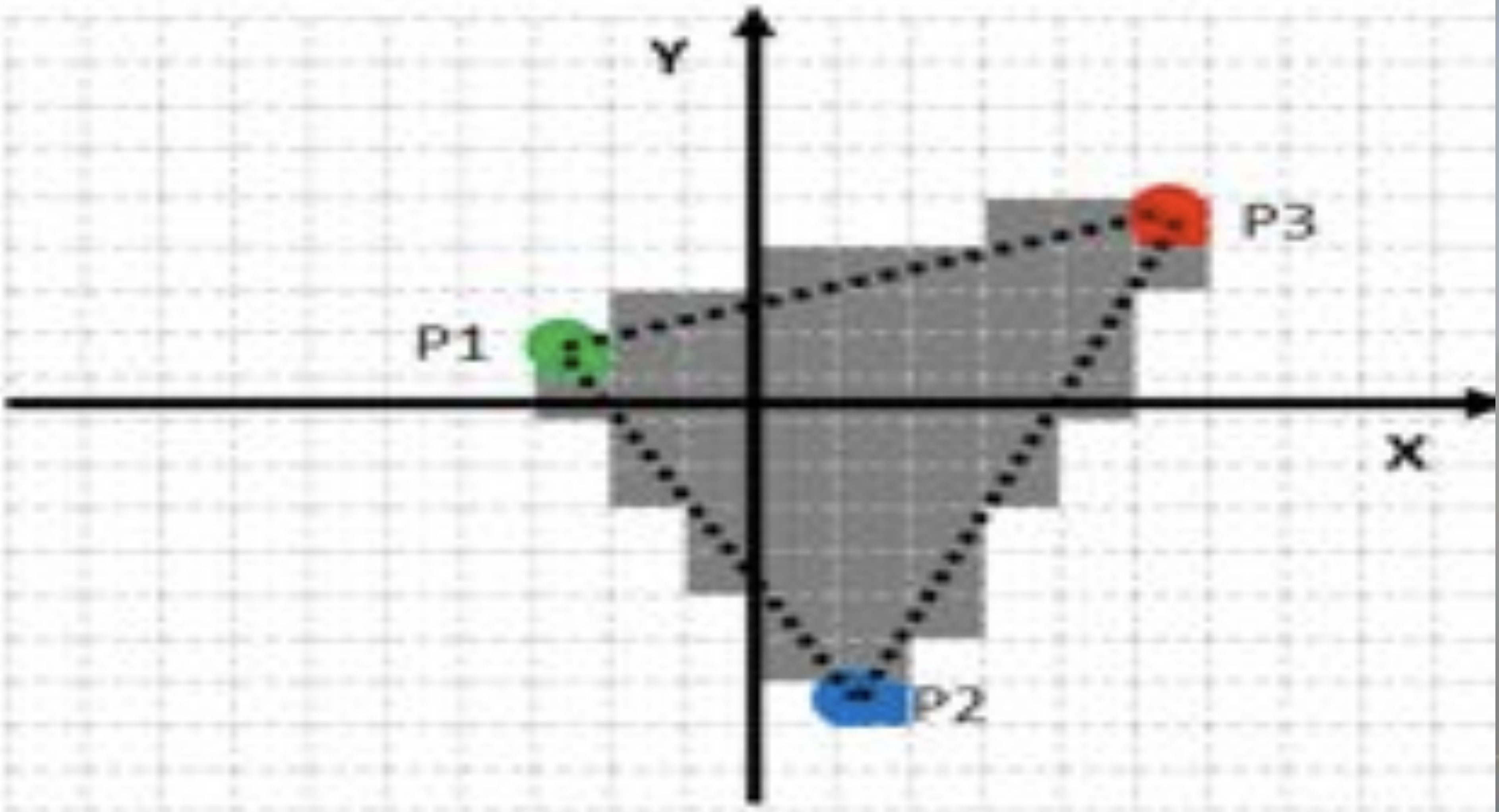
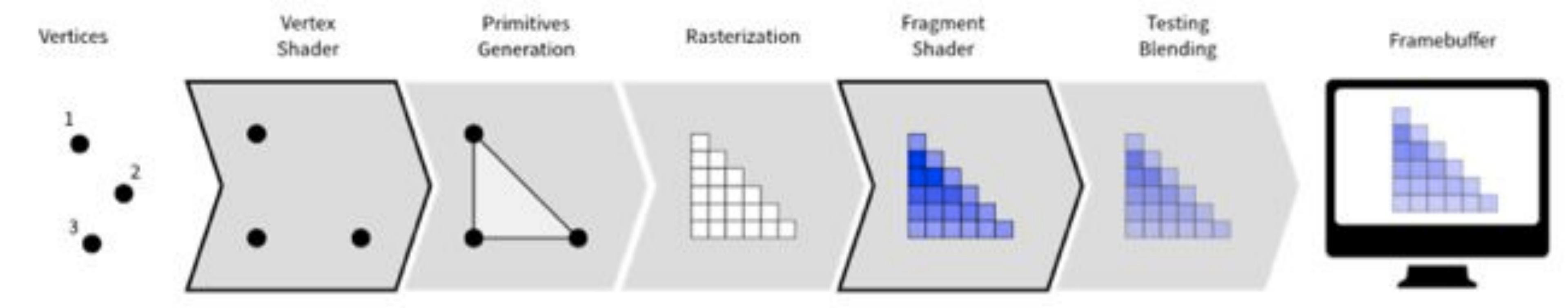
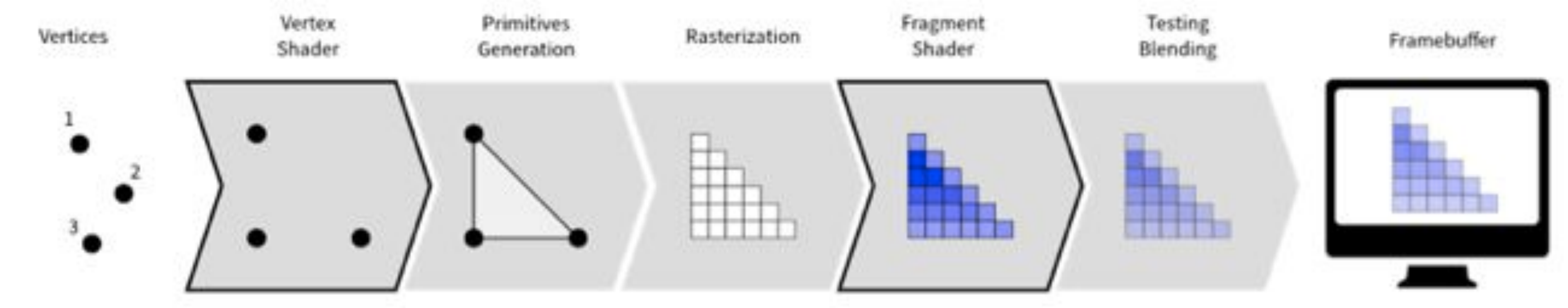


Figure 4: Rasterization of a primitive into 58 fragments

Fragment Shaders

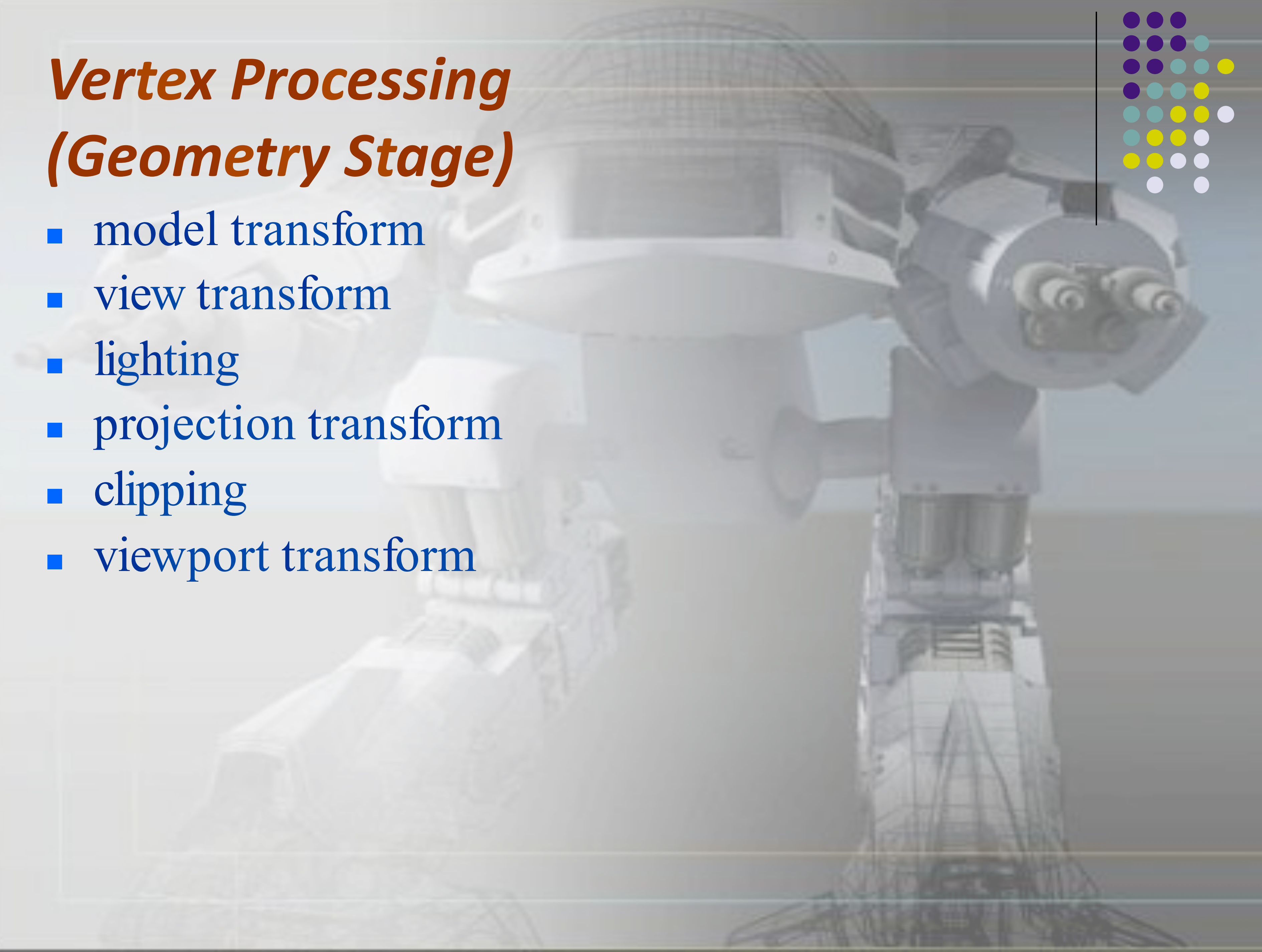


Fragment Shaders



Vertex Processing (Geometry Stage)

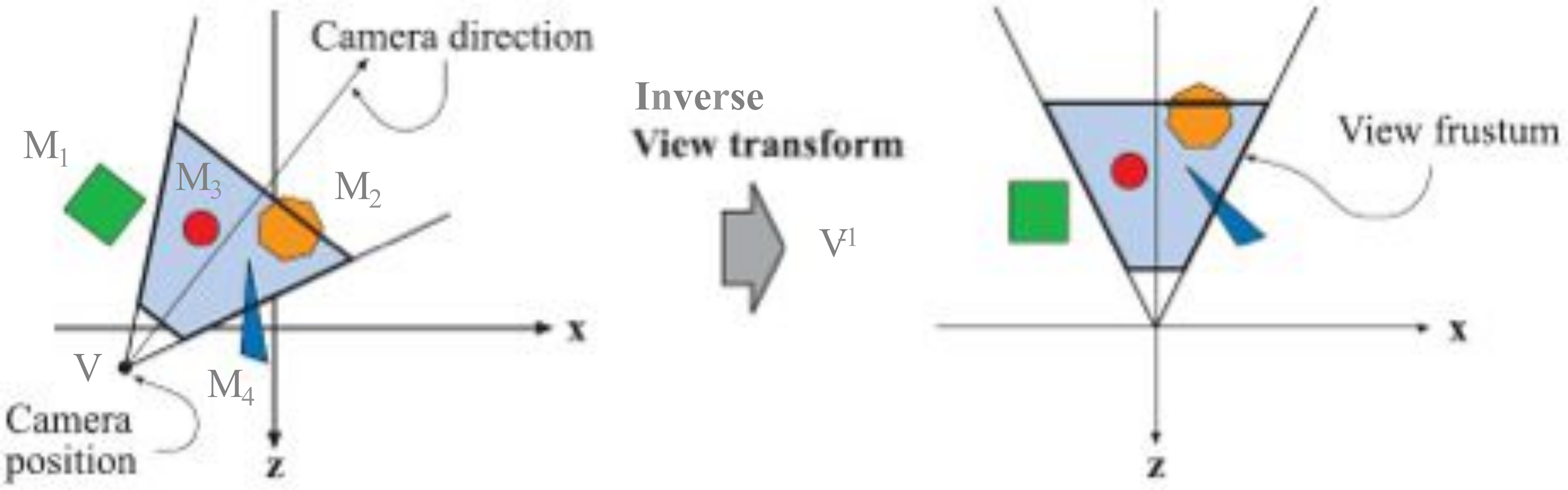
- model transform
- view transform
- lighting
- projection transform
- clipping
- viewport transform



Model Transform
View Transform

- each object and the respective vertices are positioned, oriented, scaled in the scene with a model transform
- camera is positioned and oriented, represented by the view transform
- i. e., the inverse view transform is the transform that places the camera at the origin of the coordinate system, facing in the negative z-direction
- entire scene is transformed with the inverse view transform

Model Transform
View Transform



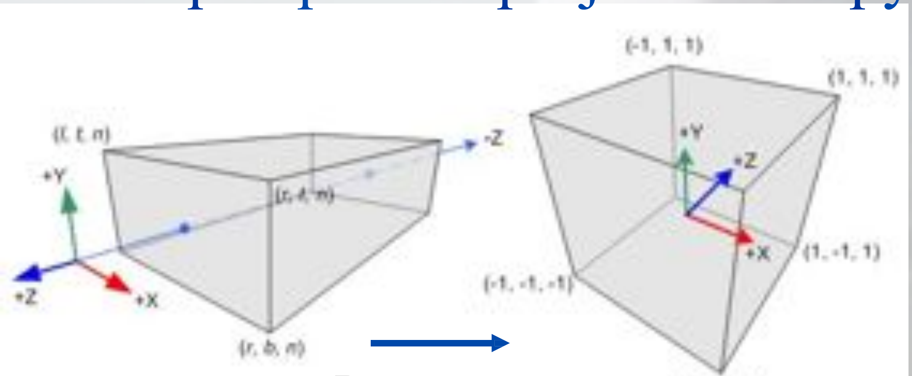
- M_1, M_2, M_3, M_4, Vare matrices representing transformations
- M_1, M_2, M_3, M_4 are model transforms to place the objects in the scene
- Vplaces and orientates the camera in space
  - V^{-1} transforms the camera to the origin looking along the negative z-axis
- model and view transforms are combined in the modelview transform
- the modelview transform $V^{-1}M_{1..4}$ is applied to the objects

Lighting

- interaction of light sources and surfaces is represented with a lighting / illumination model
- lighting computes color for each vertex
  - based on light source positions and properties
  - based on transformed position, transformed normal, and material properties of a vertex

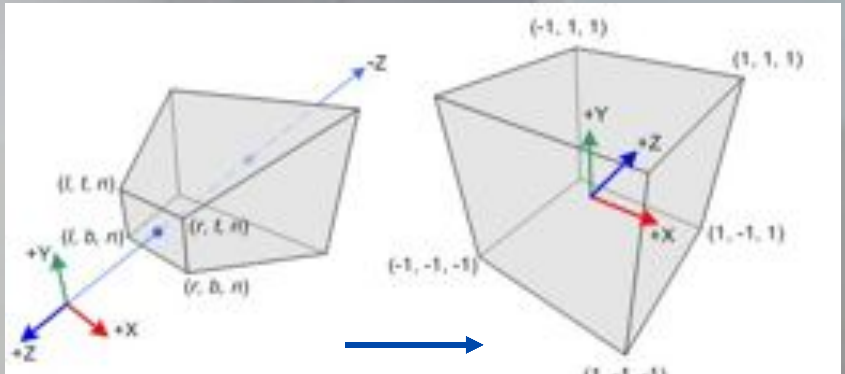
Projection Transform

- P transforms the view volume to the canonical view volume
- the view volume depends on the camera properties
  - orthographic projection ③ cuboid
  - perspective projection ③ pyramidal frustum



$P_{orthographic}$

[Song Ho Ahn]



$P_{perspective}$

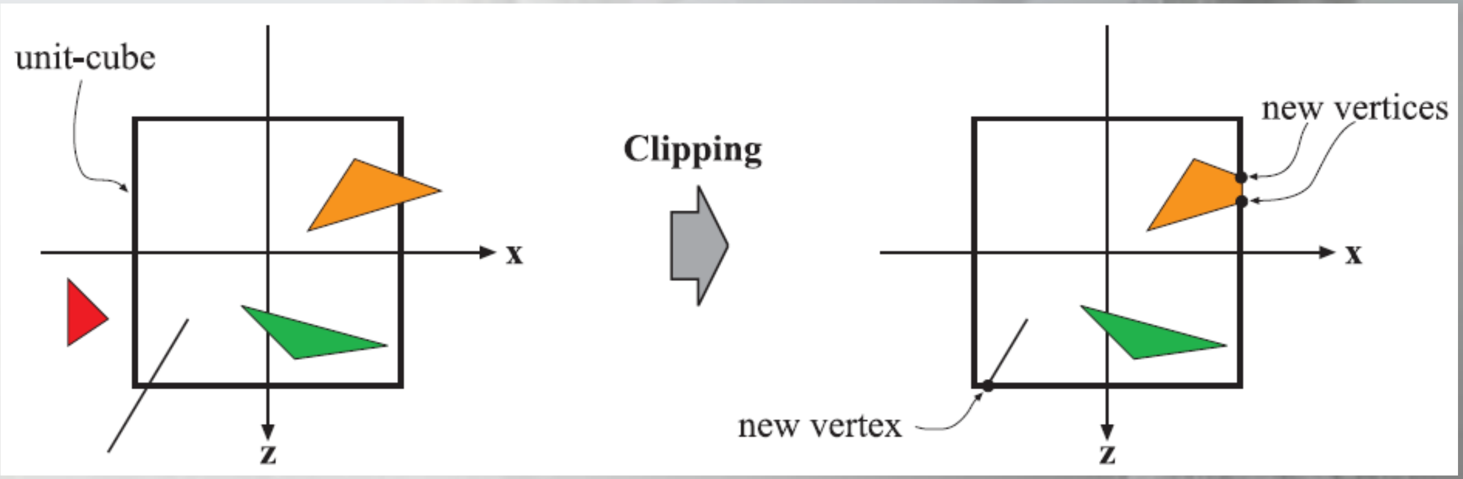
- canonical view volume is a cube from (-1,-1,-1) to (1,1,1)
- view volume is specified by near, far, left, right, bottom, top

Projection Transform

- view volume (cuboid or frustum) is transformed into a cube (canonical view volume)
- objects inside (and outside) the view volume are transformed accordingly
- orthographic
  - combination of translation and scaling
  - all objects are translated and scaled in the same way
- perspective
  - complex transformation
  - scaling factor depends on the distance of an object to the viewer
  - objects farther away from the camera appear smaller

Clipping

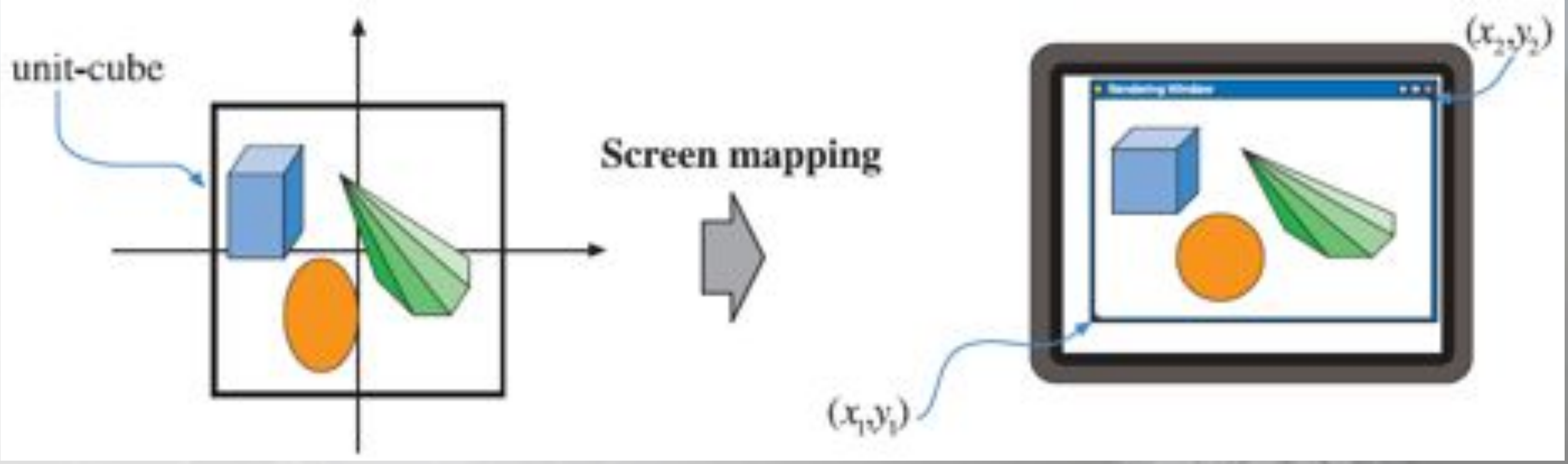
- primitives, that intersect the boundary of the view volume, are clipped
  - primitives, that are inside, are passed to the next processing stage
  - primitives, that are outside, are discarded
- clipping deletes and generates vertices and primitives



[Akenine-Moeller et al.: Real-time Rendering]

Viewport Transform / Screen Mapping

- projected primitive coordinates (x_p, y_p, z_p) are transformed to screen coordinates (x_s, y_s)
- screen coordinates together with depth value are window coordinates (x_s, y_s, z_w)



[Akenine-Moeller et al.: Real-time Rendering]

Viewport Transform / Screen Mapping

- (x_p, y_p) are translated and scaled from the range of (-1, 1) to actual pixel positions (x_s, y_s) on the display
- z_p is generally translated and scaled from the range of (-1, 1) to (0,1) for z_w
- screen coordinates (x_s, y_s) represent the pixel position of a fragment that is generated in a subsequent step
- z_w , the depth value, is an attribute of this fragment used for further processing

Vertex Processing - Summary

object space

- ➠ modelview transform

eye space / camera space

- ➠ lighting, projection

clip space / normalized device coordinates

- ➠ clipping, viewport transform

window space

Vertex Processing - Summary

- input
  - vertices in object / model space
  - 3D positions
  - attributes such as normal, material properties, texture coords
- output
  - vertices in window space
  - 2D pixel positions
  - attributes such as normal, material properties, texture coords
  - additional or updated attributes such as
    - normalized depth (distance to the viewer)
    - color (result of the evaluation of the lighting model)

Outline

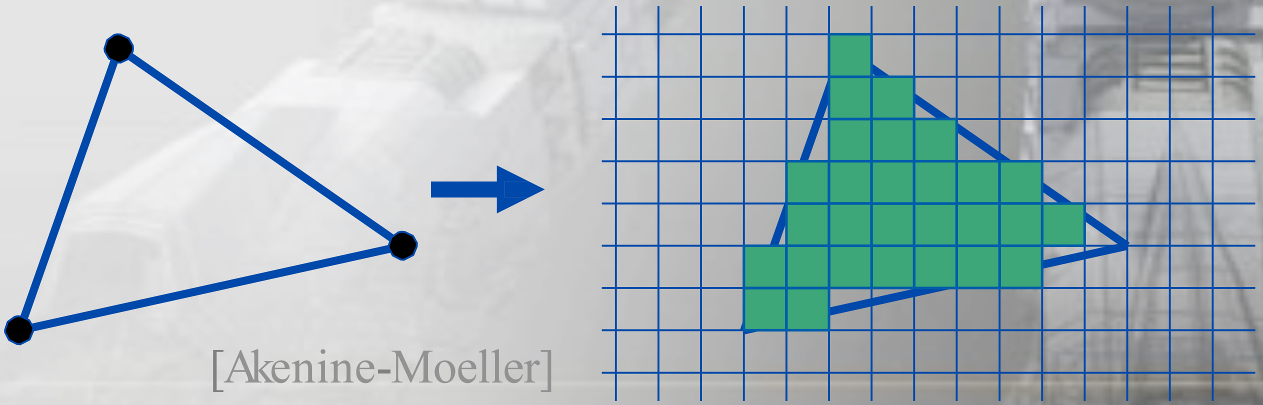
- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Primitive Assembly / Rasterization

- primitive assembly
  - vertex information and connectivity information are
  - combined for further processing of points, lines and triangles
- geometry shader
  - change, delete, generate primitives
- rasterization
  - converts primitives into fragments
  - computes positions of screen pixels
  - that are affected by a primitive
  - generates a fragment for each affected pixel position
  - interpolates attributes from vertices to fragments

Rasterization

- input
  - vertices with attributes and connectivity information
  - attributes: color, depth, texture coordinates
- output
  - fragments with attributes
    - pixel position
    - interpolated color, depth, texture coordinates



[Akenine-Moeller]

Raster operations

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Raster operations:Fragment Processing

- fragment attributes are processed and tests are performed
  - fragment attributes are processed
  - fragments are discarded or
  - fragments pass a test and finally update the framebuffer
- processing and testing make use of
  - fragment attributes
  - textures
  - additional data that is available for each pixel position
    - depth buffer, color buffer, stencil buffer, accumulation buffer

Raster operations:Attribute Processing

- texture lookup
  - use texture coords to look up a texel (pixel of a texture image)
- texturing
  - combination of color and texel
- fog
  - adaptation of color based on fog color and depth value
- antialiasing
  - adaptation of alpha value (and color)
  - color has three components: red, green, blue
  - color is represented as a 4D vector (red, green, blue, alpha)

Raster operations: Tests

- scissor test
  - check if fragment is inside a specified rectangle
  - used for, e. g., masked rendering
- alpha test
  - check if the alpha value fulfills a certain requirement
  - comparison with a specified value
  - used for, e. g., transparency and billboarding
- stencil test
  - check if the stencil value in the framebuffer at the position of the fragment fulfills a certain requirement
  - comparison with a specified value
  - used for various rendering effects, e. g. masking, shadows

Raster operations: Depth Test

- depth test
  - compare depth value of the fragment and depth value of the framebuffer at the position of the fragment
  - used for resolving the visibility
  - if the depth value of the fragment is larger than the framebuffer depth value, the fragment is discarded
  - if the depth value of the fragment is smaller than the framebuffer depth value, the fragment passes and (potentially) overwrites the current color and depth values in the framebuffer

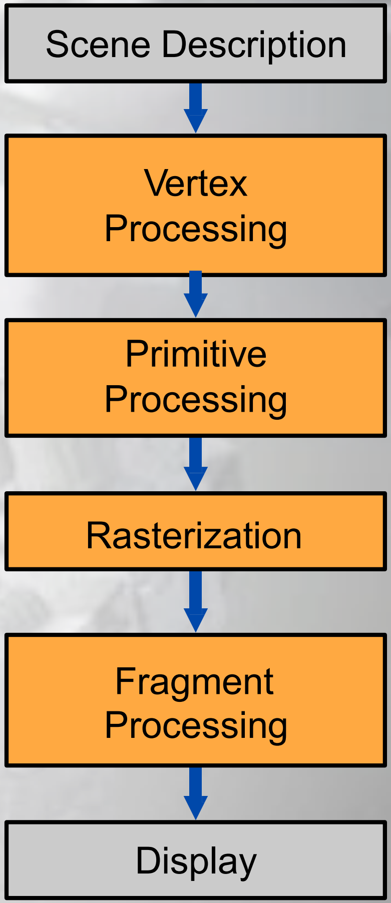
Raster operations: Blending / Merging

- blending
  - combines the fragment color with the framebuffer color at the position of the fragment
  - usually determined by the alpha values
  - resulting color (including alpha value) is used to update the framebuffer

Raster operations: Fragment Processing - Summary

- texture lookup
- texturing
- fog
- antialiasing
- scissor test
- alpha test
- stencil test
- depth test
- blending
- dithering
- logical operations

Rendering Pipeline - Summary



Rendering Pipeline - Summary

- primitives consist of vertices
- vertices have attributes (color, depth, texture coords)
- vertices are transformed and lit
- primitives are rasterized into fragments / pixel candidates with interpolated attributes
- fragments are processed using
  - their attributes such as color, depth, texture coordinates
  - texture data / image data
  - framebuffer data / data per pixel position (color, depth, stencil, accumulation)
- if a fragment passes all tests, it replaces the pixel data in the framebuffer