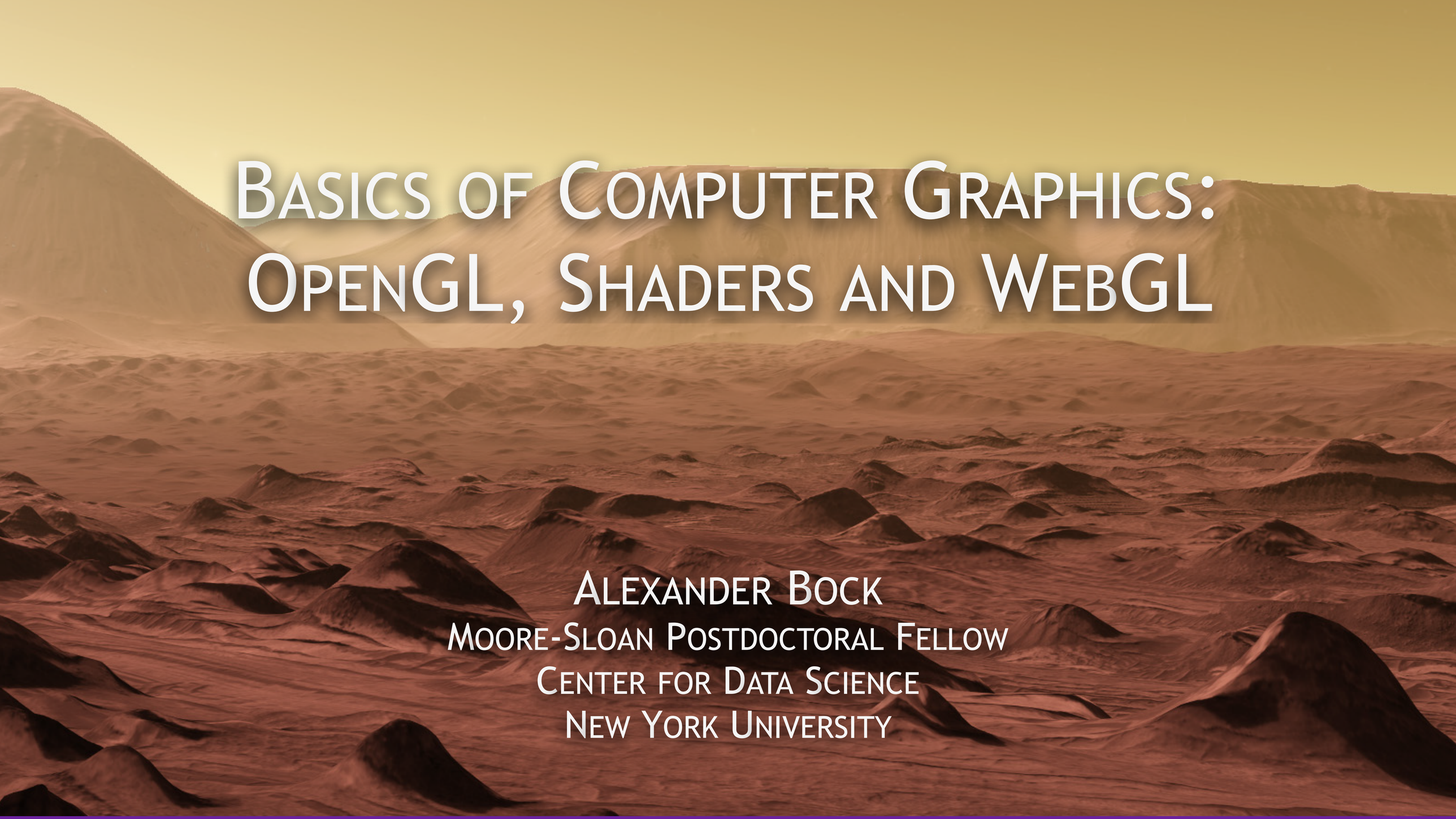
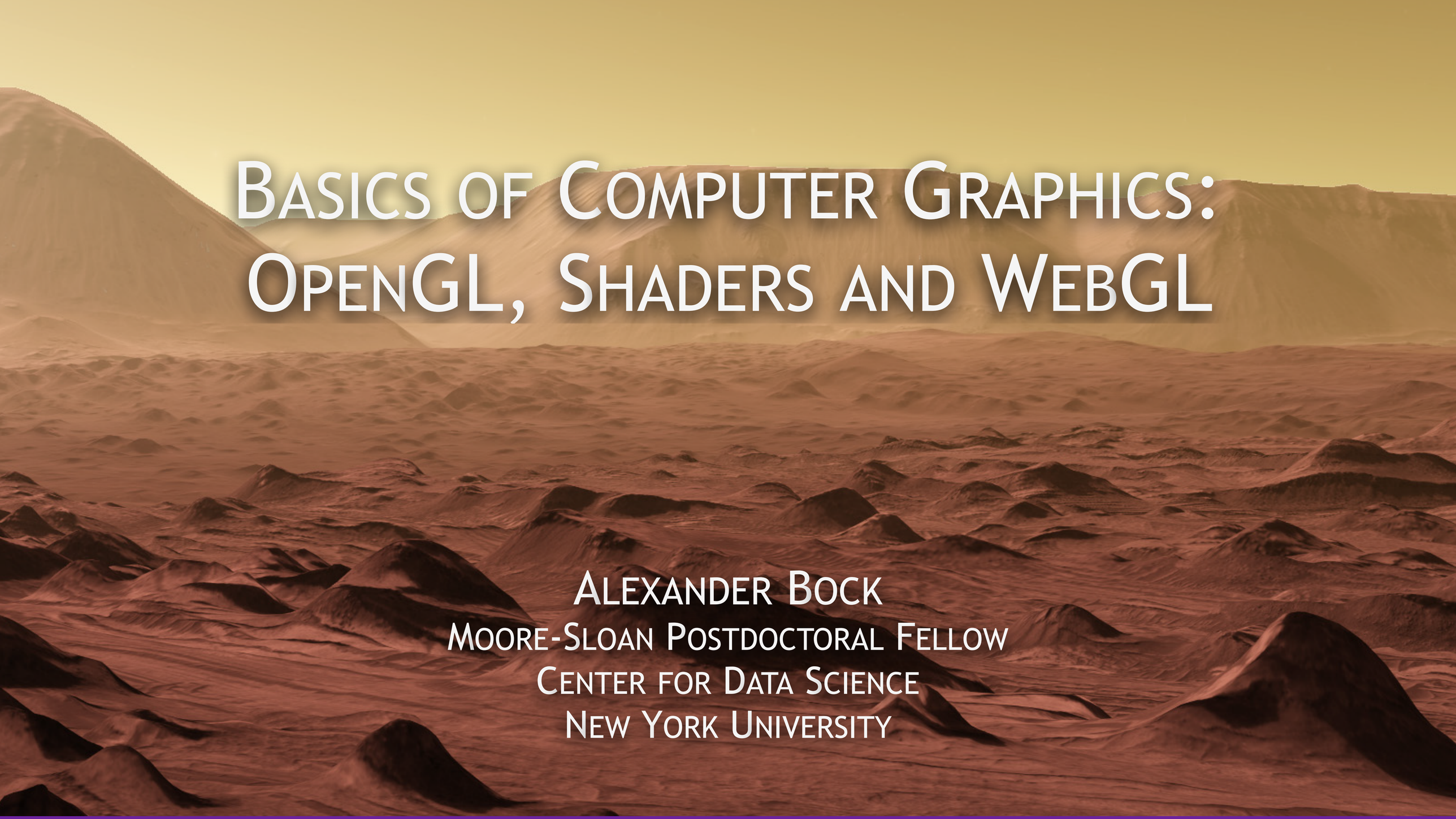


Basics of Computer Graphics: OpenGL, Shaders and WebGL

Alexander Bock
Moore-Sloan Postdoctoral Fellow
Center for Data Science
New York University

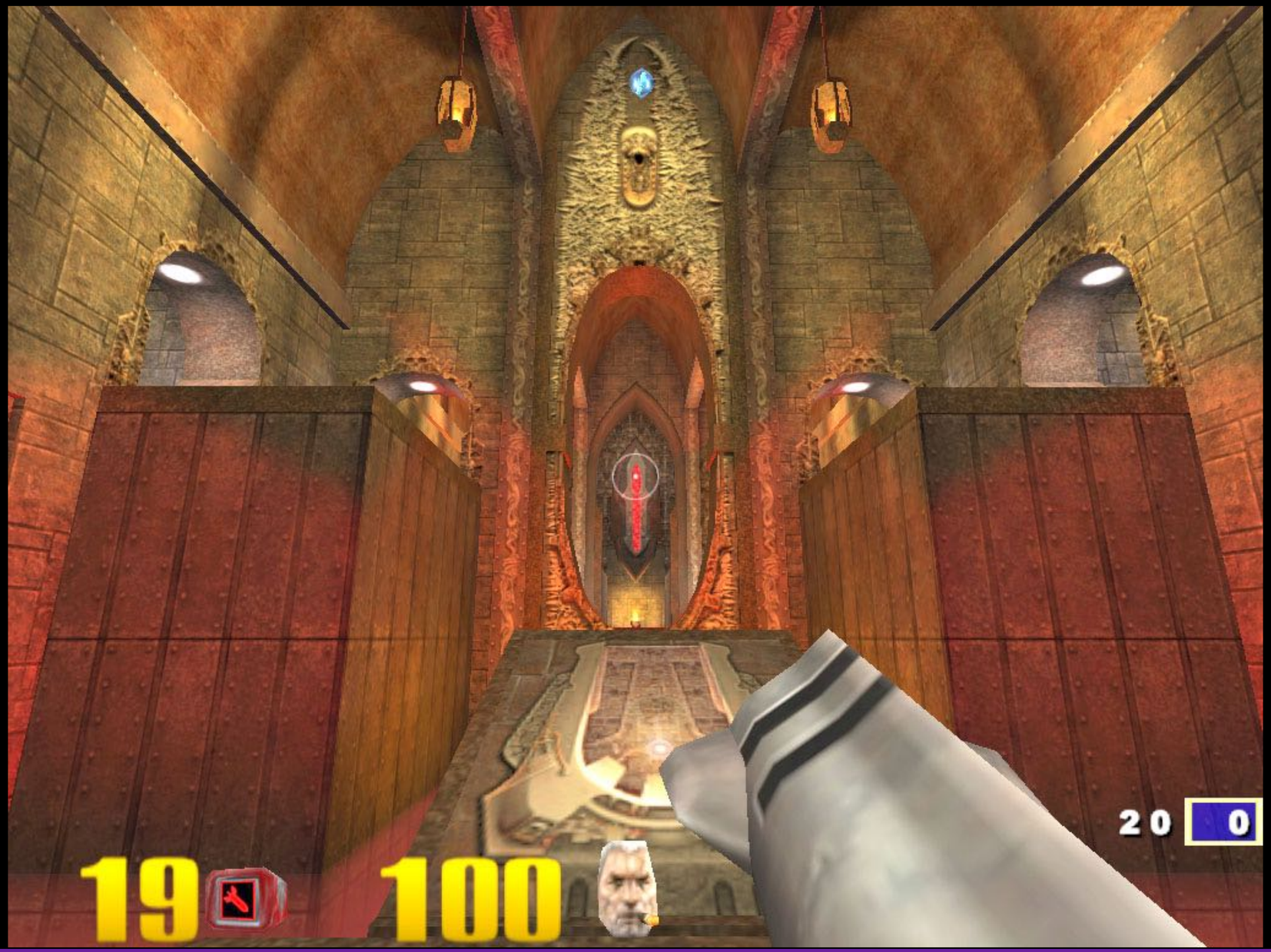


Basics of Computer Graphics: OpenGL, Shaders and WebGL

Alexander Bock
Moore-Sloan Postdoctoral Fellow
Center for Data Science
New York University

- Slides:
  - http://alexanderbock.eu/lectures/2018/ds_ga_3001_017_opengl.pdf
  - http://alexanderbock.eu/lectures/2018/ds_ga_3001_017_opengl/index.html

Why bother?



Quake 3 (2001)

Why bother?

Kingdom Come Deliverance (2018)

Why bother?

- 2008: GeForce 280
  - 240 cores
  - 933 gigaflops
- 2013: GeForce Titan
  - 2688 cores
  - 4.5 teraflops
- 2014: GeForce Titan X
  - 3072 cores
  - 7 teraflops
- 2017: GeForce Titan V
  - 5120 cores
  - 15 teraflops
  - 110 teraflops for machine learning

Why bother?

- 2008: GeForce 280
  - 240 cores
  - 933 gigaflops
- 2013: GeForce Titan
  - 2688 cores
  - 4.5 teraflops
- 2014: GeForce Titan X
  - 3072 cores
  - 7 teraflops
- 2017: GeForce Titan V
  - 5120 cores
  - 15 teraflops
  - 110 teraflops for machine learning

2005!

- IBM Blue Gene/P

What is OpenGL?

OpenGL

- API for rendering 2D and 3D graphics
- Used to communicate with a Graphics Processing Unit
- Released in 1992

- Terminology
  - OpenGL vs OpenGL ES
    - OpenGL: Desktop applications
    - OpenGL ES: Embedded systems, based on OpenGL
  - WebGL: Implementation of OpenGL ES in JavaScript

OpenGL Versions

- OpenGL
  - 1.0 (1992)
  - 2.0 (2004)
    - Programmable pipeline introduced (vertex + fragment shader)
  - 3.0 (2008)
    - Radical changes to the API (Core vs Compatibility mode)
    - More control, higher performance, less intuitive to learn
  - 4.6 (2017)
    - Current version
- WebGL
  - WebGL (based on OpenGL ES 2.0 (based on OpenGL 2.0 / OpenGL 3.0))
  - WebGL2 (based on OpenGL ES 3.0 (based on OpenGL 4.3))

Coverage

- Things we will cover
  - WebGL2 and JavaScript integration
  - Shader Programs
  - Texture handling
  - Moving data from the CPU to the GPU
  - Render loops
- Things we will not cover (in detail or at all)
  - Glue code for desktop applications (window creation, etc)
  - Fixed-function pipeline
  - 3D transformations

- Everything in the lecture is a taster; almost each slide could fill an hour's lecture; go and read about topics on your own! Topics in *italic* are concepts that you can easy search for

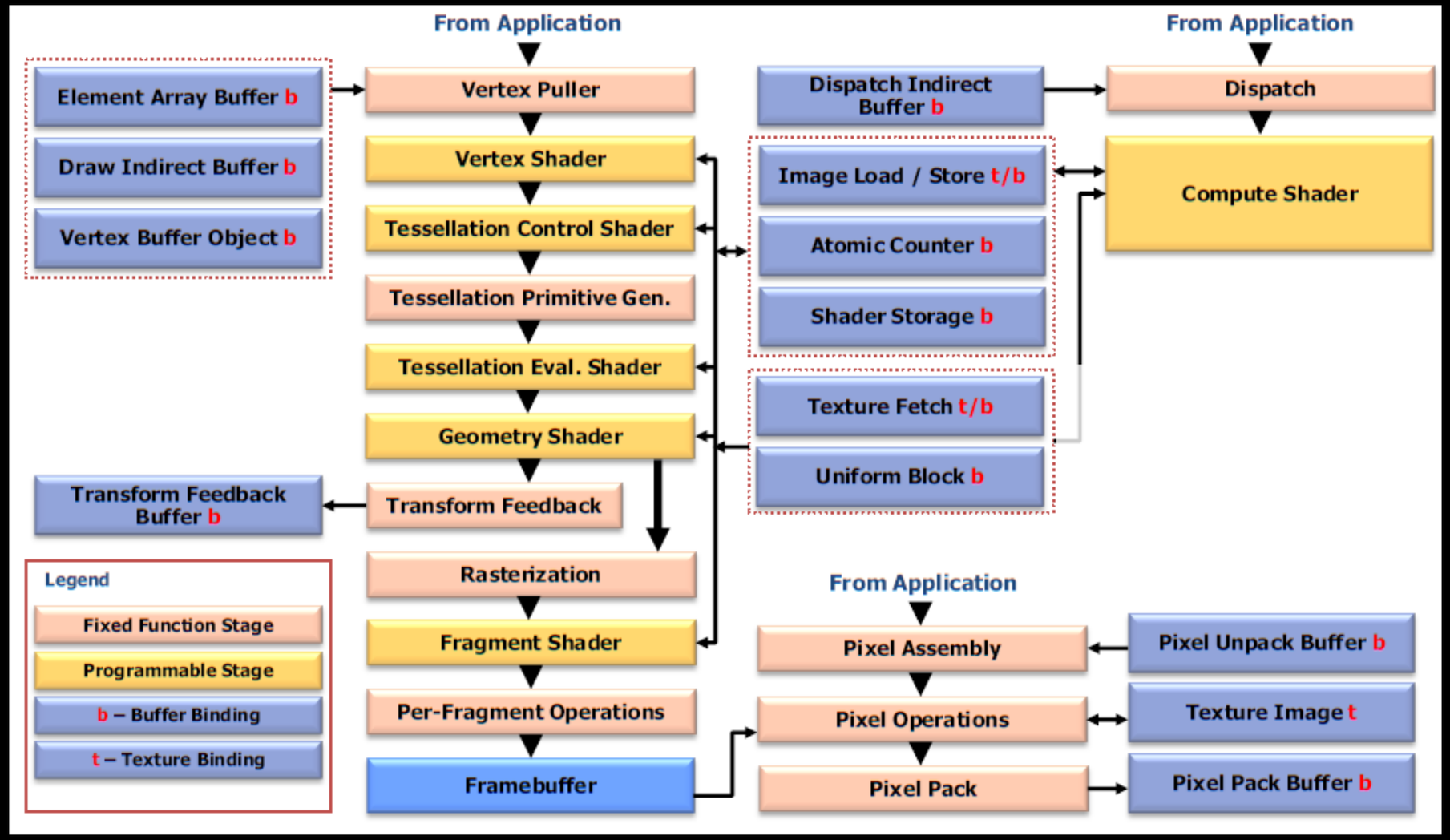
Legend

- Everything in the lecture is a taster
  - Almost each slide could fill an hour's lecture
  - -> Go and read about topics on your own!
  - Topics in *italic* are concepts that you can easy search for

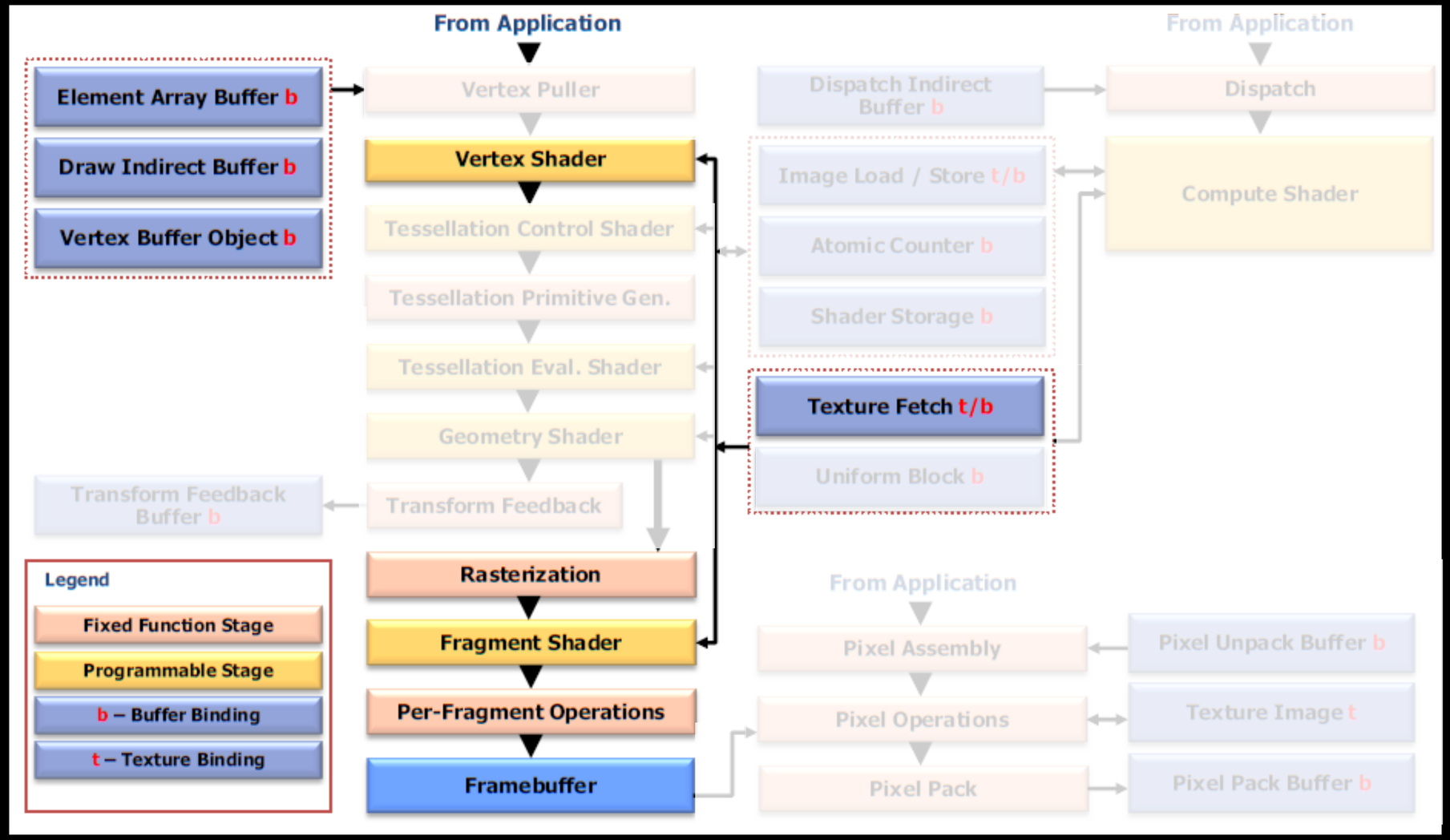
- Helvetica Neue is for text
- `Menlo is for code, constants, OpenGL functions`
- `purple are types and keywords`

Graphics Pipeline

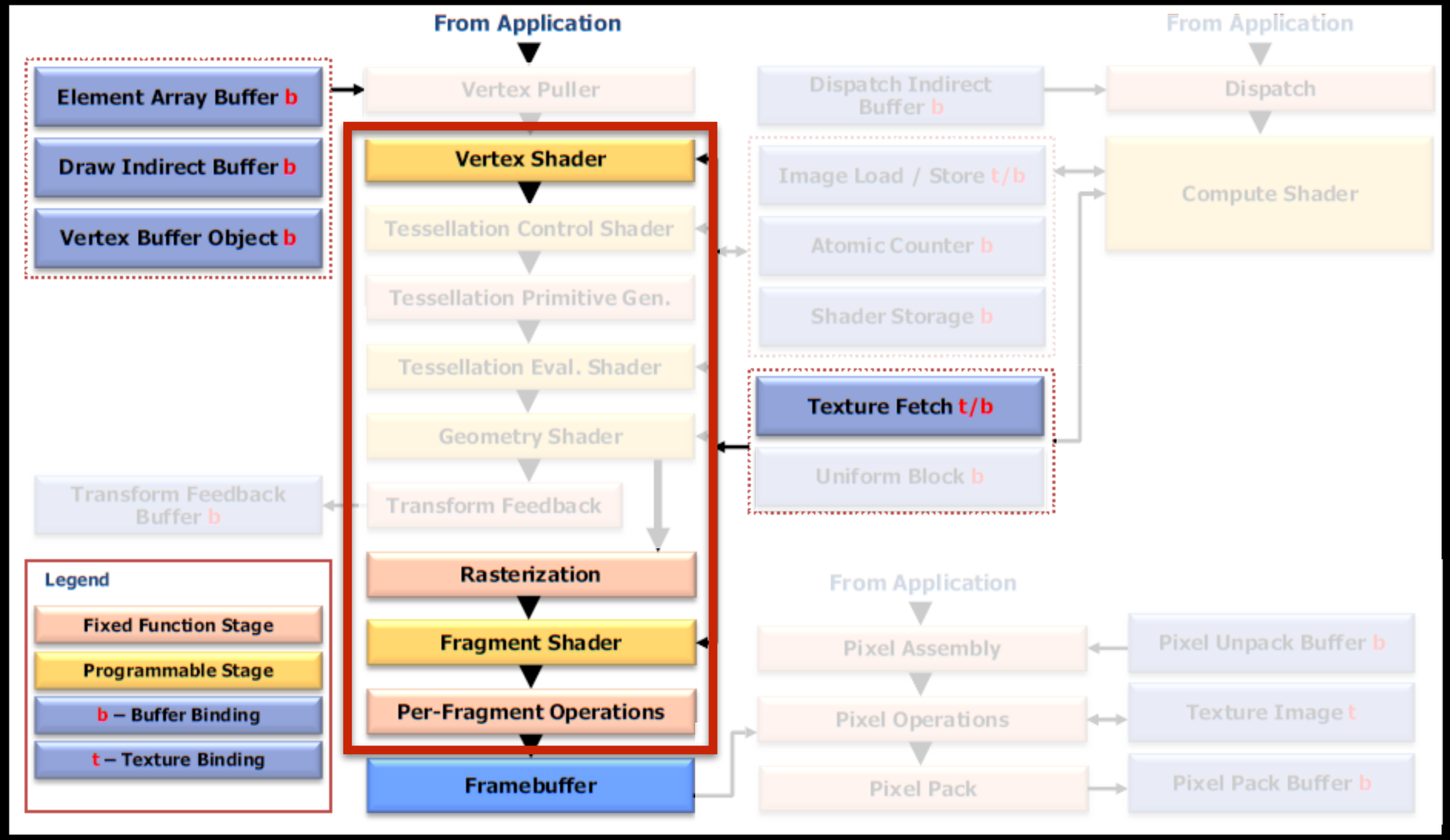
GPU Pipeline



GPU Pipeline



GPU Pipeline



GPU Pipeline - Vertex Shader

- Input: 1 Vertex
  - Single, individual vertex
  - Ordering of vertices into the shader is ***not*** defined. Regardless of how they are specified, vertices can arrive at ***any*** order!
- Output: 1 Vertex (cannot discard vertices here)

- Vertex Shader is executed exactly once **for each** vertex
- Vertices can have user-defined attributes (position, color, texture coordinates, ...) that are stored in *Vertex Buffer Objects*
- Sample usage:
  - Applying a transformation matrix (scaling, rotation, translation, shearing, ...)

GPU Pipeline - Rasterization

- Input: Primitive
- Output: List of fragments
  - Fragment (simplified): Candidate pixel with depth information. Pixels in the final image will get their color from any number of fragments.

- Performs, among others, barycentric interpolation of values over vertices
- Fixed function
  - -> No shader program is available (yet?). The only interaction is possible through state changes (for example: `glLineStipple`, `glLineWidth`, `glPointSize`)

GPU Pipeline - Fragment Shader

- Input: 1 Fragment
- Output: 0 or 1 Fragment

- Fragment shader is executed exactly one **for each** fragment regardless whether it will end up on the screen or not*
- Fragments have a screen position and depth information + other predefined attributes + user-defined attributes
- Fragments can be `discard`ed
- Sample usage:
  - Per-pixel lighting
  - Volume rendering
  - Texturing
  - ...

GPU Pipeline - Per-Fragment Operations

- Fixed pipeline
- Can discard or merge fragments
- Assembling multiple fragments (*MSAA*, Multi-sampling antialiasing)
- Depth Buffer Test
  - Result depending on `GL_DEPTH_TEST` and glDepthFunc
- Blending
  - Result depending on `GL_BLEND` and glBlendFunc
- ...

GPU Pipeline - Other Shaders

- Tessellation Control Shader
  - Determines the level of tessellation for a particular patch (group of vertices)
  - Filter vertices
- Tessellation Evaluation Shader
  - Generates a single new vertex for a patch per execution
- Geometry Shader
  - Takes a single primitive (e.g. triangle) and outputs zero or more primitives of the same or different types
- Compute Shader
  - Performs abstract, non graphics-related programs

GPU Pipeline - Program Objects

- Individual shaders are combined into *Programs*
  - Shaders are *attached* to programs
  - Multiple shaders are allowed, only one of them may have a `main` method
- Shaders need to be compiled
- Programs need to be linked

- Workflow
  1. Create shaders
  2. Compile shaders
  3. Attach shaders
  4. Link program

OpenGL / WebGL

OpenGL

- OpenGL is a state machine
  - Settings are stored internally by the OpenGL
  - Current state is used when vertices are pushed through the pipeline
  - Almost everything is a state (`glClearColor`, `glPointSize`, `glLineWidth`, `glBlendFunc`, active shaders, antialiasing, viewport, ...)

- Primitives
  - WebGL2
    - `POINTS`, `LINE_STRIP`, `LINE_LOOP`, `LINES`, `TRIANGLE_STRIP`, `TRIANGLE_FAN`, `TRIANGLES`

Observations

- All OpenGL methods are prefixed by `gl` (on Desktop machines, different for every implementation)
- Most methods have a suffix that declares the type of arguments
  - Methods are of type
    `gl.<function>{ε 1 2 3 4}{ε b s i i64 f d ub us ui ui64}{ε v}`
  - 1234: Number of arguments
  - ‘**b**yte’ ‘**s**hort’ ‘**i**nt’ ‘**i**nt **64** bit’ ‘**f**loat’ ‘**d**ouble’ ‘**u**nsigned byte’ ‘**u**nsigned short’ ‘**u**nsigned int’ ‘**u**nsigned int 64 bit’
  - **v**ector

- Examples:
  - `gl.uniformMatrix4fv` (function: uniformMatrix, four arguments of type float, passed as a vector)
  - `gl.getVertexAttribiv` (function: getVertexAttrib, 1 argument of type integer, passed as a vector)

Depth Testing

- `gl.enable(gl.DEPTH_TEST)` / `gl.disable(gl.DEPTH_TEST)`
- `gl.depthFunc(...)`
- Default: `gl.disable(gl.DEPTH_TEST)`, `gl.depthFunc(gl.ALWAYS)`
- If depth testing is disabled, later fragments will overwrite the values in the frame buffer (Painter's Algorithm)
- `gl.depthFunc` specifies when fragments pass the per-fragment test of the pipeline
- Possible values:
  - `NEVER`, `LESS`, `LEQUAL`, `GREATER`, `NOTEQUAL`, `GEQUAL`, `ALWAYS`

Blending

- `gl.enable(gl.BLEND)` / `gl.disable(gl.BLEND)`
- `gl.blendFunc`(source, destination)
- Default: `gl.disable(GL_BLEND)`, `gl.blendFunc(GL_ONE, GL_ZERO)`
- Determines how the incoming fragment (source) is blended with the value already written in the frame buffer (destination) using linear interpolation
- Possible values (among others):
  - `ZERO`, `ONE`, `SRC_COLOR`, `ONE_MINUS_SRC_COLOR`, `DST_COLOR`, `ONE_MINUS_DST_COLOR`, `SRC_ALPHA`, `ONE_MINUS_SRC_ALPHA`, `DST_ALPHA`, `ONE_MINUS_DST_ALPHA`

Blending

- `gl.enable(gl.BLEND)` / `gl.disable(gl.BLEND)`
- `gl.blendFunc`(source, destination)
- Default: `gl.disable(GL_BLEND)`, `gl.blendFunc(GL_ONE, GL_ZERO)`
- Determines how the incoming fragment (source) is blended with the value already written in the frame buffer (destination) using linear interpolation
- Possible values (among others):
  - `ZERO`, `ONE`, `SRC_COLOR`, `ONE_MINUS_SRC_COLOR`, `DST_COLOR`, `ONE_MINUS_DST_COLOR`, `SRC_ALPHA`, `ONE_MINUS_SRC_ALPHA`, `DST_ALPHA`, `ONE_MINUS_DST_ALPHA`

- `Danger Zone: Incoming order of fragments is **not** defined!`

OpenGL Shading Language (GLSL)

OpenGL Shading Language

- Imperative, “C-like” programming language
- Shaders are programs that run in parallel on the GPU
  - GeForce Titan V: 5120 cores
- Start in `main() { ... }`
- We do not have:
  - No pointer chasing, pointer arithmetic (`*(p + 2)`) (sort-of)
  - No dynamic arrays (no memory allocation on a heap)
  - Automatic type conversion (in general)
- We do have:
  - Additional in-built types for managing vectors, matrices, textures, ...
  - Speed

New Data Types - Vectors

- `{ε b d i u}vec{2 3 4}`
- ‘**b**ool’ ‘**d**ouble’ ‘**i**nteger’ ‘**u**nsigned integer’
- {2 3 4}: number of components
- Examples
  - `vec2`: two float components
  - `dvec4`: four double components
  - …

```
vec4 pos = vec4(1, 2, 3, 4);
```

New Data Types - Vectors

- Swizzling
  - Implemented in hardware -> almost free
- Three identical variants of accessing components as long as sets are not mixed
  - `xyzw`
  - `rgba`
  - `stpq`

```
vec4 pos = vec4(1, 2, 3, 4);
float c1 = pos.x;
float c2 = pos.w;
vec2  c3 = pos.xy;
vec2  c4 = pos.xz;
vec4  c5 = pos.wyzx;
vec4  c6 = pos.zzxx;
```

New Data Types - Vectors

- Arithmetic operations work component-wise
- Built-in functions operate on vectors component-wise
  - For example: `cos`, `sin`, `abs`, `sqrt`, ...
  - Also available `length`, `distance`, `normalize`, `dot`, `cross`, ...

```
vec2 a = vec2(13, 37);
vec2 b = vec2(85, 19);
vec2(a.x * b.x, a.y * b.y) == a * b;
```

```
vec2 a = vec2(13, 37);
vec3 b = vec3(85, 19, 08);
vec2(a.x * b.x, a.y * b.y) == a * b.xy
```

New Data Types - Matrices

- `{ε d}mat{2 3 4}{ε x2 x3 x4}`
- Examples:
  - `mat2` (= `mat2x2`): float, 2 columns, 2 rows
  - `dmat3` (= `dmat3x3`): double, 3 columns, 3 rows
  - `mat3x4`: float, 3 columns, 4 rows
- Matrices (on default) are **column-major** (but can be changed)
- Accessors:

```
mat3x4 matrix;
vec3 col1 = matrix[0];  // First column
float val1 = matrix[2][1]; // Third column, second row
float val2 = matrix[2].x; // val2 == val1
```

- Arithmetic operations behave as expected
  - `matαxβ * vecβ   = vecβ`
  - `matαxβ * matαxβ = matαxβ`
  - `vecα * matβxδ` compile error

Textures

- Textures
  - 1D, 2D, 3D images associated with a sampling function
  - Linear sampling, (bi- / tri-)linear interpolation, anisotropic interpolation,
- In GLSL: opaque type `sampler1D`, `sampler2D`, `sampler3D` (among others)
- Access using built-in functions:
  - `vec4 texture(sampler1D texture, float texture_coordinate);`
  - `vec4 texture(sampler2D texture, vec2  texture_coordinate);`
  - `vec4 texture(sampler3D texture, vec3  texture_coordinate);`

New Qualifiers

- `uniform`
  - A qualifier for a global variable, which can be set from the host program using GL functions (gl.uniformX)
  - These values are used to define behaviour between shader invocation instances
  - Example: `uniform float brightness;`
- `in`/`out`
  - Three meanings
    1. Denote values that are passed from one shader stage to the next. Have to be (in general) named the same in both shader stages or explicitly numbered. These values are interpolated between vertices.
    2. `in` Names vertex attributes in the vertex shader (e.g. position, normal, texture coordinates, ...)
    3. `out` Names output values for the fragment shader (99% of the time a color value `vec4`)
  - Example:
    - Vertex Shader: `out vec3 normal;`
    - Fragment Shader: `in vec3 normal;`

Pre-Defined Variables

- Vertex shader (In)
  `gl_VertexID`
  `gl_InstanceID`
  ...

- Vertex shader (Out)
  `gl_Position` (vertex position)
  `gl_PointSize`
  ...

- Fragment shader (In)
  `gl_FragCoord` (fragment position)
  `gl_FrontFacing`
  `gl_PointCoord`
  ...

- Fragment shader (Out)
  `gl_FragDepth` (fragment depth)
  ...

JavaScript

JavaScript

- Dealing with WebGL2
  - Based on OpenGL ES 3.0
  - Biggest compatibility with modern OpenGL for Desktops

- Useful libraries
  - https://github.com/toji/gl-matrix
    - Provides easy matrix operations on the JavaScript side
  - https://github.com/frenchtoast747/webgl-obj-loader
    - Library for loading OBJ models

JavaScript Integration

```
<html>
<script type="text/javascript">
function main() {
  var canvas = document.querySelector("#glCanvas");
  var gl = canvas.getContext("webgl2");
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);
}
</script>

<body onload="main();">
 <canvas id="glCanvas" width="640" height="480"></canvas>
</body>
</html>
```

Example

https://github.com/alexanderbock/webgl-examples

Matrices

Transformations

- Transformations in 3D (rotation, scale, shear) can be expressed as 3x3 matrix multiplications
- Translations cannot
- *Homogeneous coordinates*
  - 3x3 -> 4x4
  - Translation by x,y,z ->

```
1 0 0 x       a         a + x
0 1 0 y   *   b    =    b + y
0 0 1 z       c         c + z
0 0 0 1       1           1       = w
```

  - Division by `w` (*perspective division*) to retrieve inhomogeneous point
- The *Model matrix* defined the result of a number of transformations that are unique for an object

View & Projection Matrix

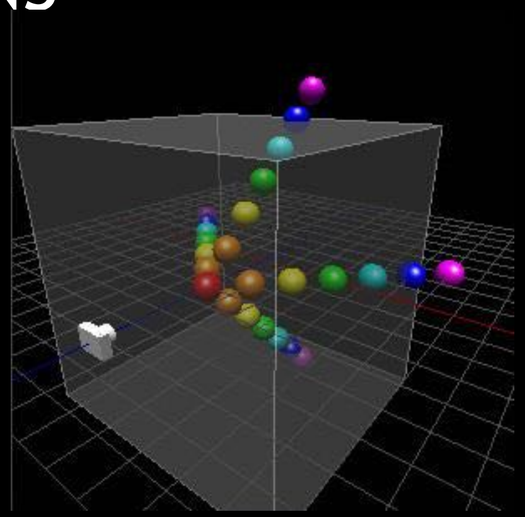
- Per default: OpenGL's camera is at (0,0,0) looking towards the -z direction
- Everything in *normalized device coordinates* (NDC) in range [-1,1]
- View matrix modifies the location of the OpenGL's camera

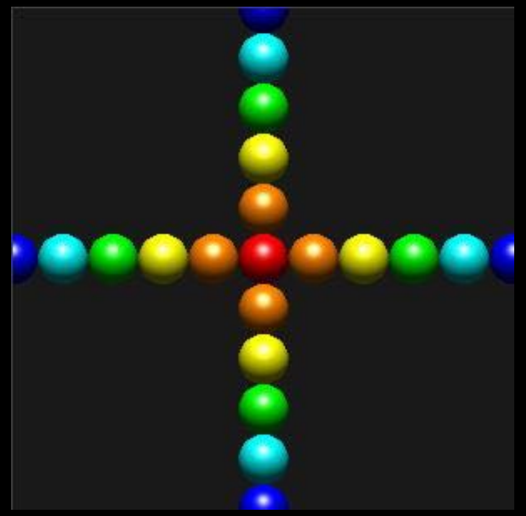
- Projection Matrix
  - Projects a 3D scene onto the 2D rendering surface
  - Projection methods
    - Orthographic projection (parallel lines remain parallel)
    - Perspective projection (parallel lines converge)

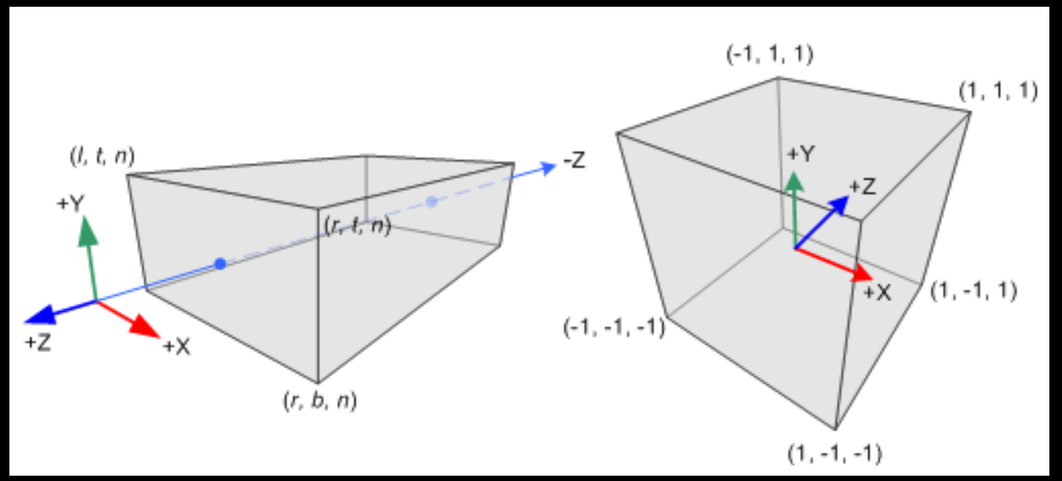
- Each vertex `v` is modified by model `M`, view `V`, and projection `P` matrices:
  `x = P * V * M * v`

Projections

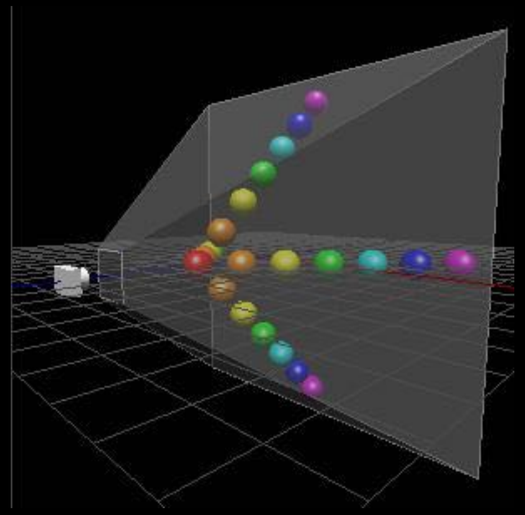
Orthographic projection

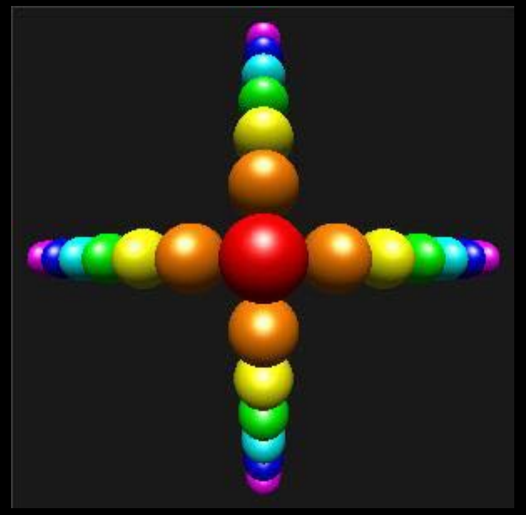


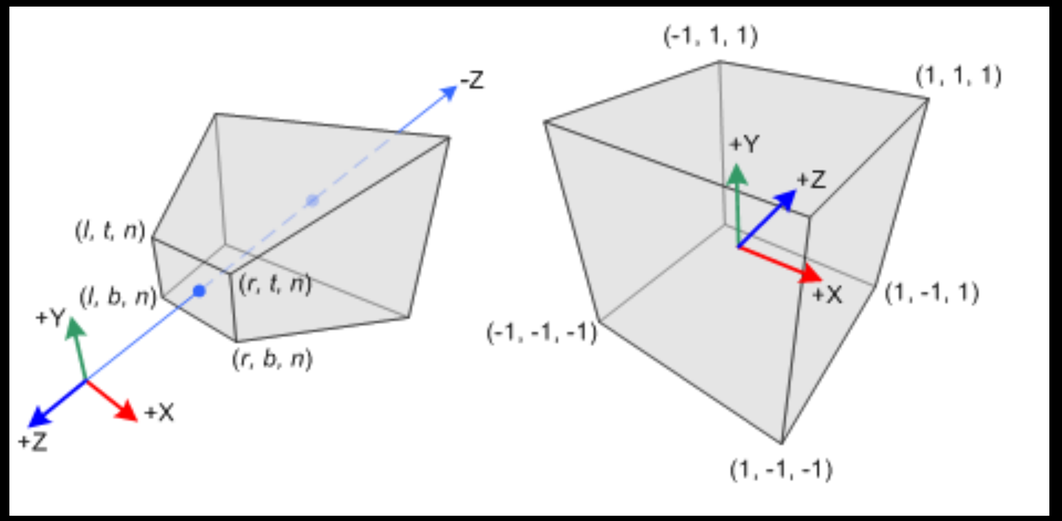




Perspective projection







[Song Ho Ahn]
"OpenGL", http://www.songho.ca

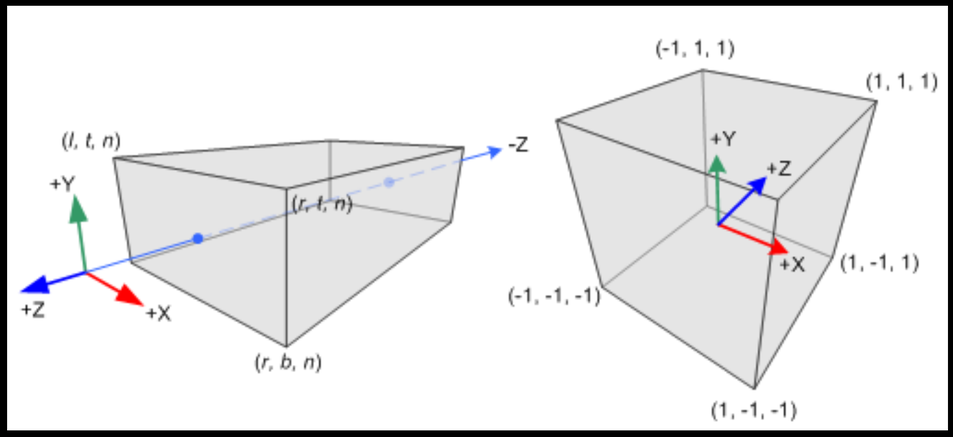
Orthographic Projection

- P =

$$\begin{bmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & -2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

for r = right, l=left, u=up, d=down, n=near, f=far planes defining the cube that is of interest

- w-component is unchanged, not requiring a perspective division
- -> The size of objects does not depend on their distance to the camera



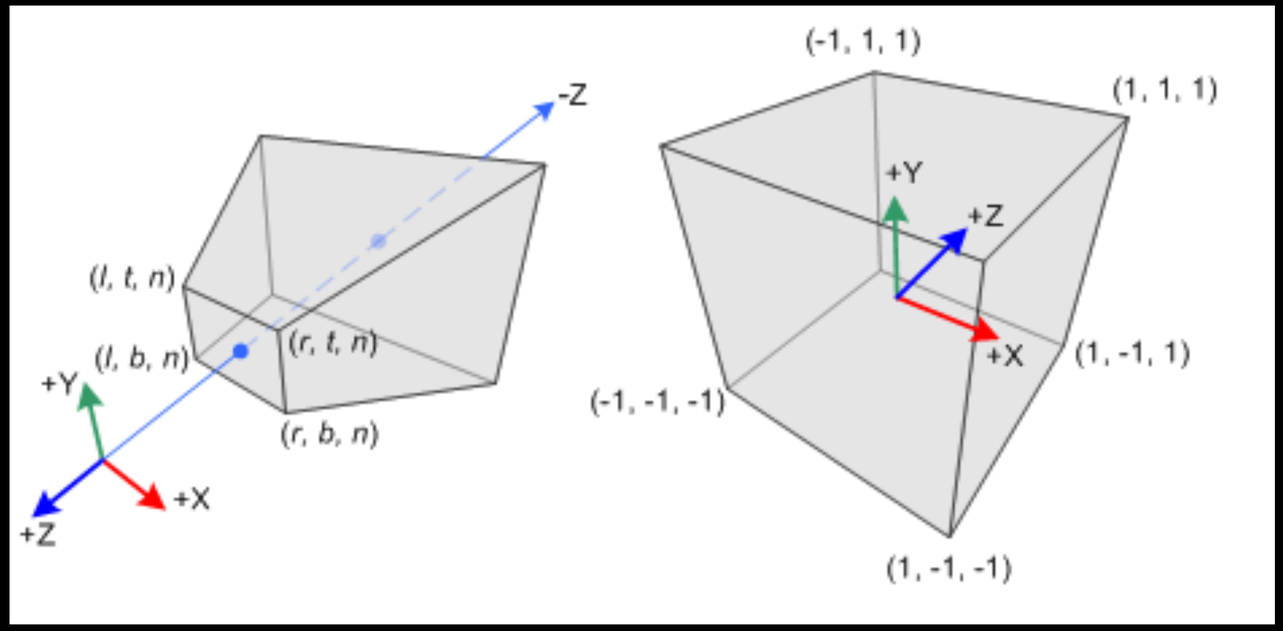
Shadowrun Returns (2013)

Perspective Projection

- P =

$$\begin{matrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{matrix}$$

for r = right, l=left, u=up, d=down, n=near, f=far planes defining the cube that is of interest



Perspective Projection

- Perspective projection matrices are often specified as field of view (fov) and aspect ratios instead:

$$P = \begin{bmatrix} f/a & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & (f+n)/(n-f) & 2fn/(n-f) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

for f = cotangent(fov / 2), a = aspect ratio


The Hunt (2018?)

Shading

Phong Shading

- Non-physically based way to create view-dependent hightlights
- For each surface point, uses the angle between the direction towards the eye (V), reflected direction (R) of the light direction (L) about the surface normal (N)
- $$\vec{R} = 2\left(\vec{L}\cdot\vec{N}\right)\vec{N} - \vec{L}$$

  (all vectors are assumed normalized)

- Then, the light information is given by:

$$I = k_a + k_d\left(\vec{L}\cdot\vec{N}\right) + k_s\left(\vec{R}\cdot\vec{V}\right)^{\alpha}$$

  with constants for ambient (k_a), diffuse (k_d), and specular (k_s) lighting

- α determines the level (=size) of specular highlights

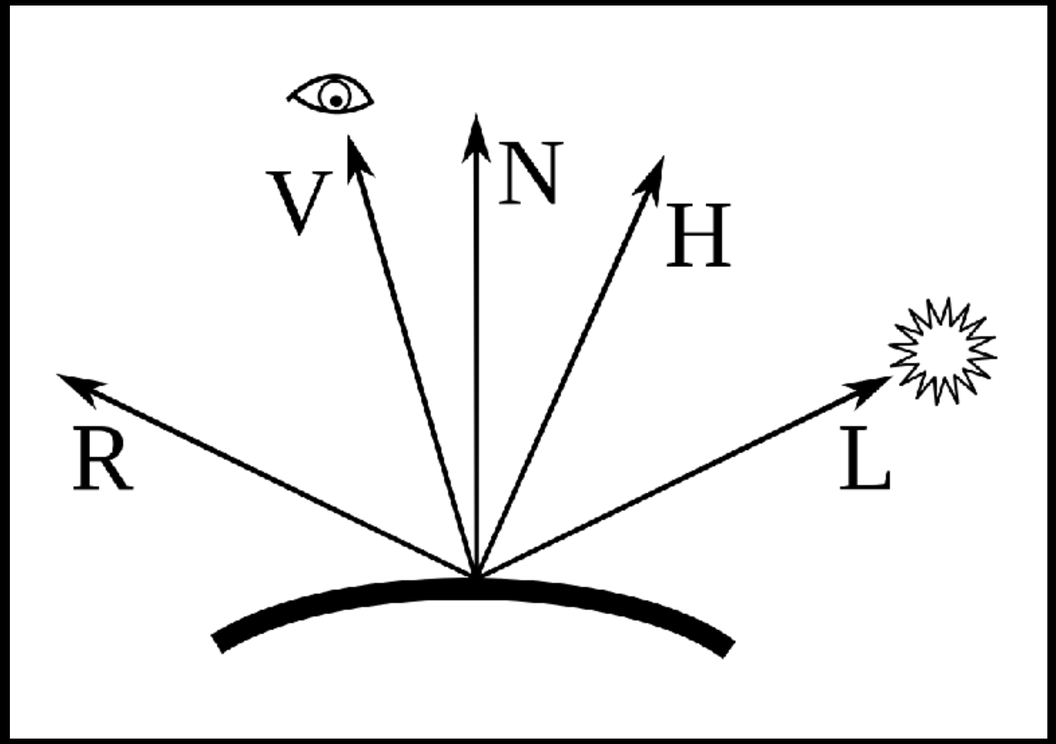


Image Copyright: Martin Kraus

Blinn-Phong Shading

- In Phong shading L has to be reflected about N every frame for each position in order to compute R * V
- Blinn-Phone shading replaces R * V with N * H, where H is the half-way vector:

$$\vec{H} = \frac{\vec{L} + \vec{V}}{||\vec{L} + \vec{V}||}$$

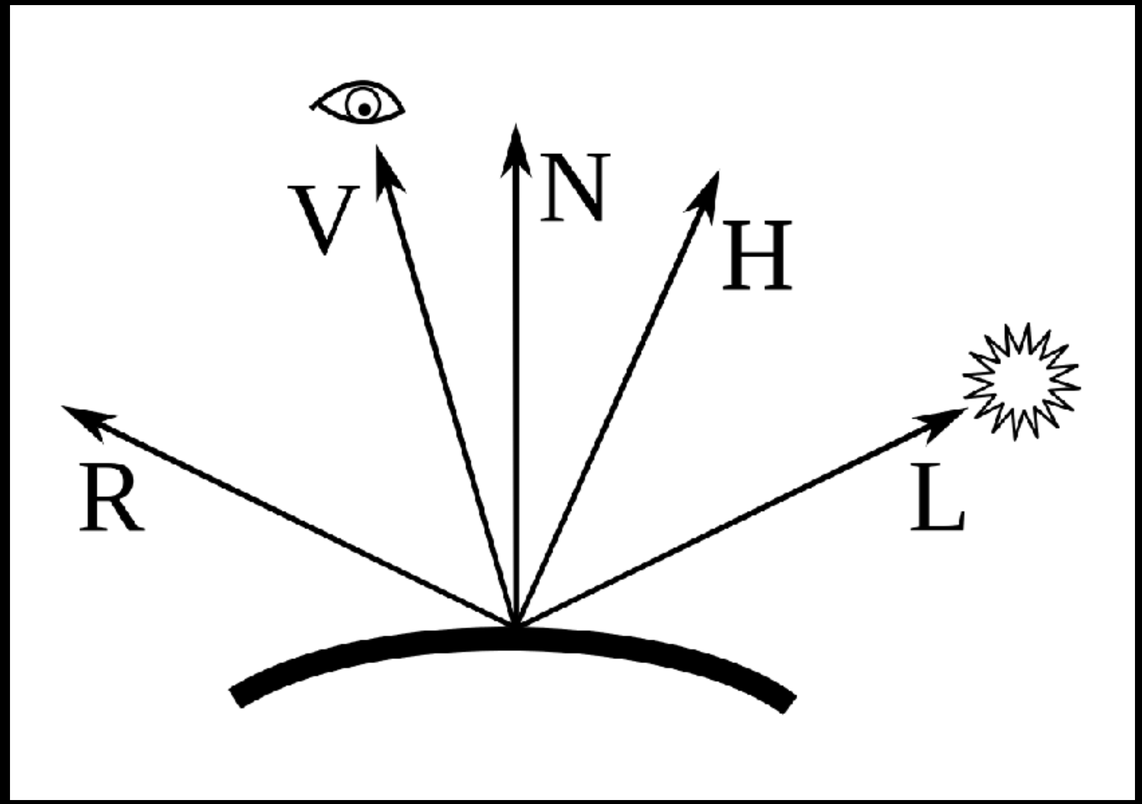


Image Copyright: Martin Kraus