Realtime Computer Graphics on GPUs

Speedup Techniques, Other APIs

Jan Kolomazník

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Computer Graphics Charles University

Optimizations Intro

Performance Bottlenecks I

- Most of the applications require steady framerate – What can slow down rendering?
- Too much geometry rendered
  - CPU/GPU can process only limited amount of data per second
  - Render only what is visible and with adequate details
- Lighting/shading computation
  - Use simpler material model
  - Limit number of generated fragments
- Data transfers between CPU/GPU
  - Try to reuse/cache data on GPU
  - Use async transfers

Performance Bottlenecks II

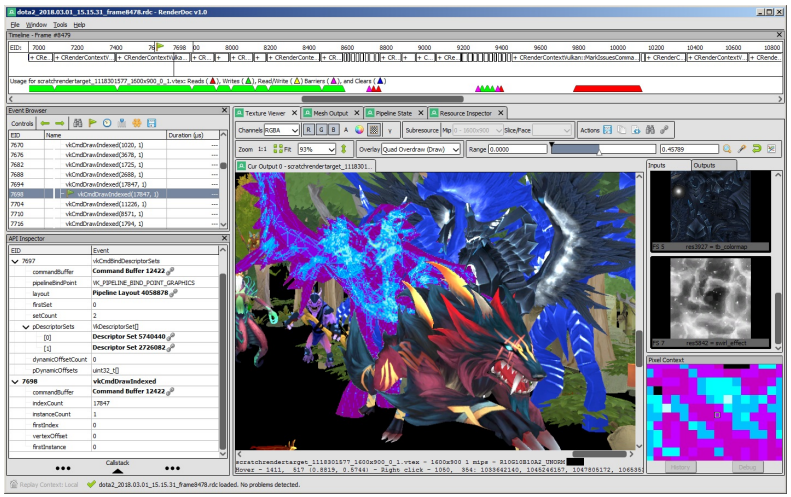
- State changes
  - Bundle object by materials
  - Use UBOs (uniform buffer objects)
- GPU idling – cannot generate work fast enough
  - Multithreaded task generation
  - Not everything must be done in every frame – reuse information (temporal consistency)
- CPU/Driver hotspots
  - Bindless textures
  - Instanced rendering
  - Indirect rendering

Different Needs

- Large open world
  - Rendering lots of objects
  - Not so many details needed for distant sections
- Indoors scenes
  - Often lots of same objects – instaced rendering
  - Only small portion of the scene visible – occluded by walls, ...
- CAD, molecular biology, ...
  - All geometry visible at once
  - Lots of self-occlusions
  - Switching levels of detail

PROFILING

- Three rules of profiling:
  - Measure!
  - Measure!
  - Measure!
- How to measure?
  - Code instrumentation
  - Sampling profilers
  - HW event counters
- Profiling influences performance
- Tools for GPU:
  - NVIDIA Nsight
  - Radeon GPU Profiler
  - RenderDoc
  - apitrace
  - CodeXL



Standard Rendering Procedure

1. Analyze world view
2. Transfer data on GPU
   - Upload/reuse geometry in VBOs
   - Upload/reuse textures
3. Issue draw commands

Scene Graph Overview

- Graphical data structure
- Manages rendering, collision, culling
- Optimizes performance in 3D environments
- Hierarchical parent-child relationships

Boundary Volume Hierarchies (BVH)

- Tree structure for geometric objects
- Nodes encapsulate object subsets (bounding volumes)
- Collision detection, ray tracing optimization
- Minimizes pairwise intersection tests
- Accelerates hit testing in ray tracing engines

Spatial Partitioning Techniques

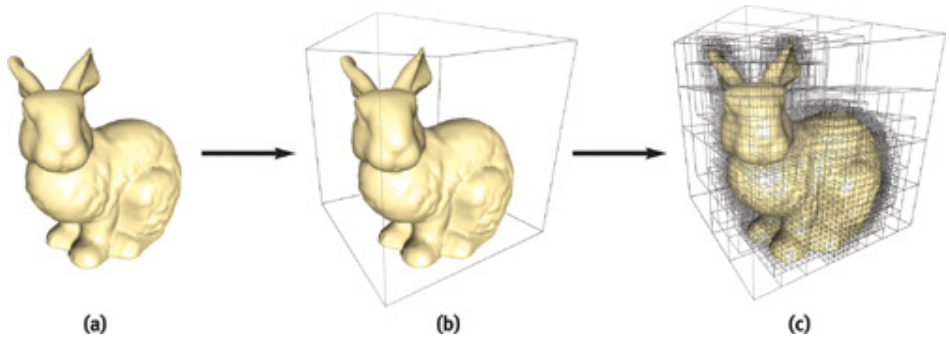
- Divides space for efficient object management
- Reduces rendering, collision complexity
- Key in visibility, physics calculations
- Enables efficient frustum culling and dynamic loading

Binary Space Partitioning (BSP)

- Recursively divides space using planes
- Render order optimization
- Efficient in visibility determination, scene traversal
- Core of early 3D video games (e.g., Doom, Quake engines)
- Used for constructing PVS (Potentially Visible Set)

Quadtrees and Octrees

- Quadtree: 2D space division into four
- Octree: 3D space division into eight
- Manage dynamic objects, scene culling
- Efficient for spatial querying of large datasets (point clouds)
- Example: Octrees in Minecraft for landscape management



Grid Partitioning

- Space divided into regular grid cells
- Simplifies spatial querying
- Used in uniform scene distributions, physics grids
- Common in particle physics simulations
- Example: Fluid dynamics where local interactions are key

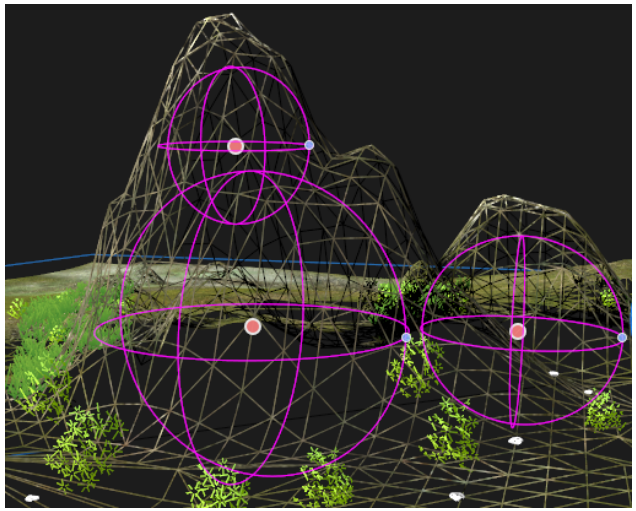
META-OBJECTS IN SCENE GRAPHS

- Occluders: Blocks non-visible objects
- Collision Geometry: Simplified object meshes for physics
- Reduces rendering and physics overhead
- Example: Occluders in urban simulation for hiding unseen buildings

Optimize Rendering

Occlusion Culling

- Do not render objects hidden behind others
- Helper objects – occluders
- CPU processing
  - Analyze scene graph + occluders to filter rendered geometry
- GPU processing
  - Z-buffer pre-render
  - Render occluders to Z-buffer
  - Occlusion queries
  - Temporal consistency – Z-buffer reprojection



Query Objects

- How to use:

```
glGenQueries(GLsizei n, GLuint * ids);
glBeginQuery(GLenum target, GLuint id);
   // ...
glEndQuery(GLenum target);
```

- Example queries:

  `GL_SAMPLES_PASSED`

  `GL_ANY_SAMPLES_PASSED`

  `GL_PRIMITIVES_GENERATED`

  `...`

- Usualy used with one frame lag

Conditional Rendering

- GL version is 3.0 or greater
- Render cheap object
- Use occlusion query to see if any of it is visible
- If it isn't, skip rendering of expensive object

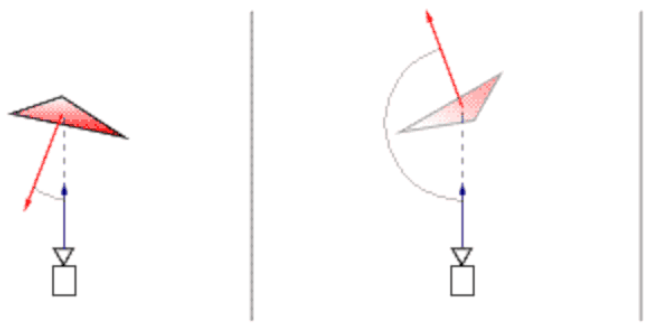
```
glBeginConditionalRender(GLuint id, GLenum mode);
glEndConditionalRender();
```

Z-buffer Optimizations

- Early Z-test:
  - Fragment shader cannot modify depth value
  - Front-to-back rendering (rough sort)
  - Double speed Z-only:
    - First pass depth (stencil) only, write z-buffer – faster when no color buffer present
    - Second pass color writes, z-buffer read-only
  - Optimizations:
    - Z-pass for major occluders
- Deffered shading

Backface Culling

- Good closed meshes
  - No need for triangle back-side rasterization
- GPU can filter (face cull) according to vertex order:
  - glEnable(GL_CULL_FACE);
  - glFrontFace(GL_CCW);
  - glCullFace(GL_BACK); // draw front faces only



Clipping Planes

- Objects outside of the view frustum are not rendered
- Set far-plane reasonable close (also increases z-buffer precision)
- Harsh end of all geometry
  - Render background geometry
  - Hide the far plane in fog (glFog())
  - More sophisticated effect in fragment shader/deffered shading

Level of Detail (LoD)

- Optimal rendering efficiency:
  - details of distant objects (pixel size) need not be drawn
  - the closest objects (and/or user focus) need the best available visual quality
- dynamic level of detail
  - rendering system adjusts individual rendering accuracy
  - global parameter definition (e.g. total approximate number of drawn triangles)
  - advance data preparation: discrete LoD levels / continuous LoD pre-processing..

Discrete LoD

- fixed LoD levels are prepared in advance
  - shape approximations with different accuracy
  - can come from the finest model – generalization (can be time consuming)
- rendering – choosing appropriate LoD level:
  - according to object-camera distance
  - according to bounding object projection size or even exact object projection – errors perpendicular to viewing direction are most noticeable
  - object importance, *player focus*, ..
  - global balancing (declared number of triangles to draw)

Continuous LoD

- Progressive meshes (Hoppe 1996)
  - Hierarchy of simplifying operations
  - Smooth transition between differetn LoDs
  - High memory consumption
  - Problematic to implement on parallel architectures (GPUs)
- LoD streaming:
  - Analyze required LoD levels
  - Stream missing geometry and required operations to GPU (prioritize)
  - Temporal coherence

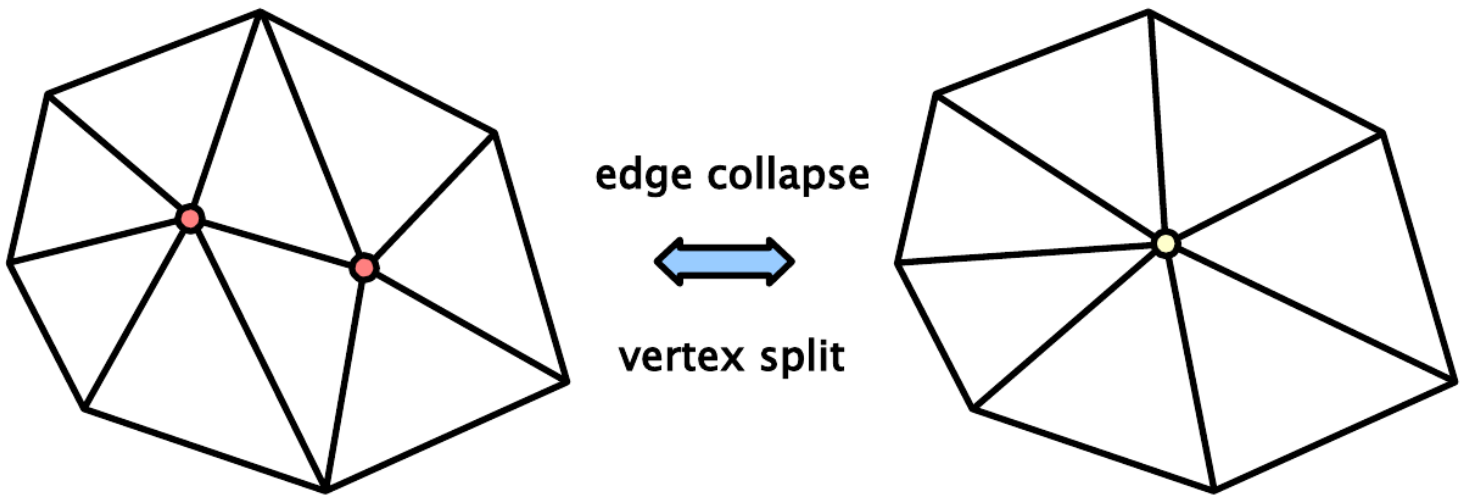
LoD Transitions

- **Simple Switching** – Directly switches between different LoD models.
  - **Hysteresis Implementation:**
    - Prevents frequent switching (reduces ”popping”).
    - Utilizes a threshold buffer around transition points.
- **Blending Neighboring LoD Levels**
  - Smooth transition by blending models of adjacent LoDs.
  - **Dual Rendering:**
    - Render both levels with partial transparency.
    - Blend based on distance or viewer focus.
  - **Linear Combination:**
    - Use linear interpolation between LoDs.
  - **Opacity Management:**
    - Current LoD level is rendered opaque.
    - New LoD level is rendered semitransparent on top.
    - Z-writing enabled for the current LoD level to maintain depth integrity.

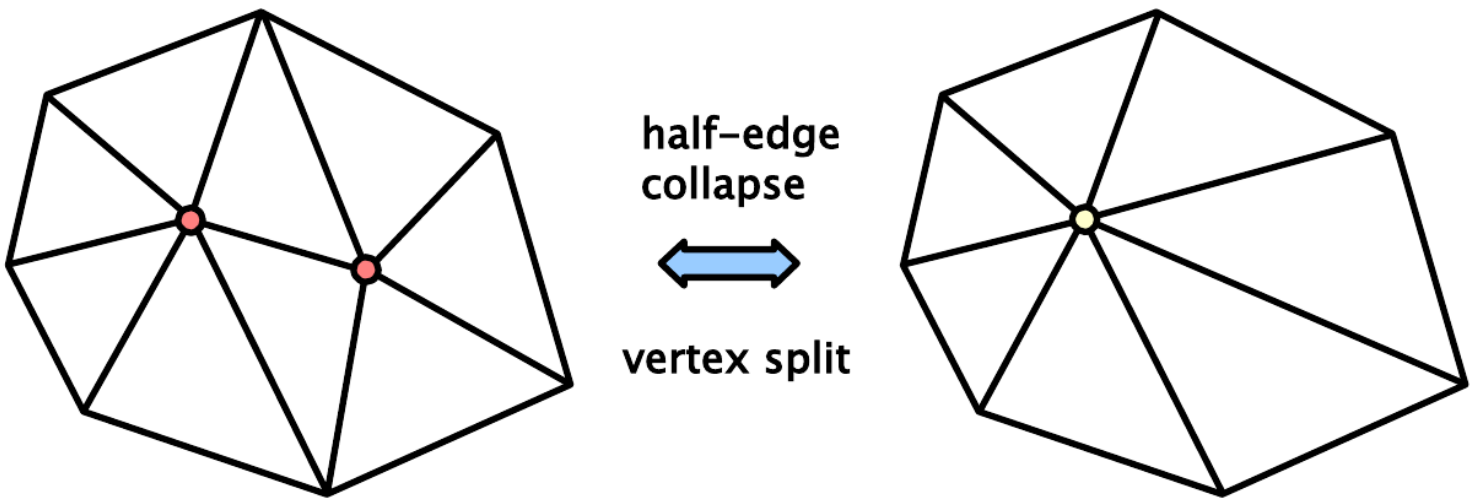
Generating LoDs

- top-down approach
  - based on the simplest shape representation (low-poly), additional details are introduced
  - less frequent (subdivision surfaces..)
- bottom-up
  - starts with detailed (most accurate) model
  - gradual simplification / generalization (data reduction)

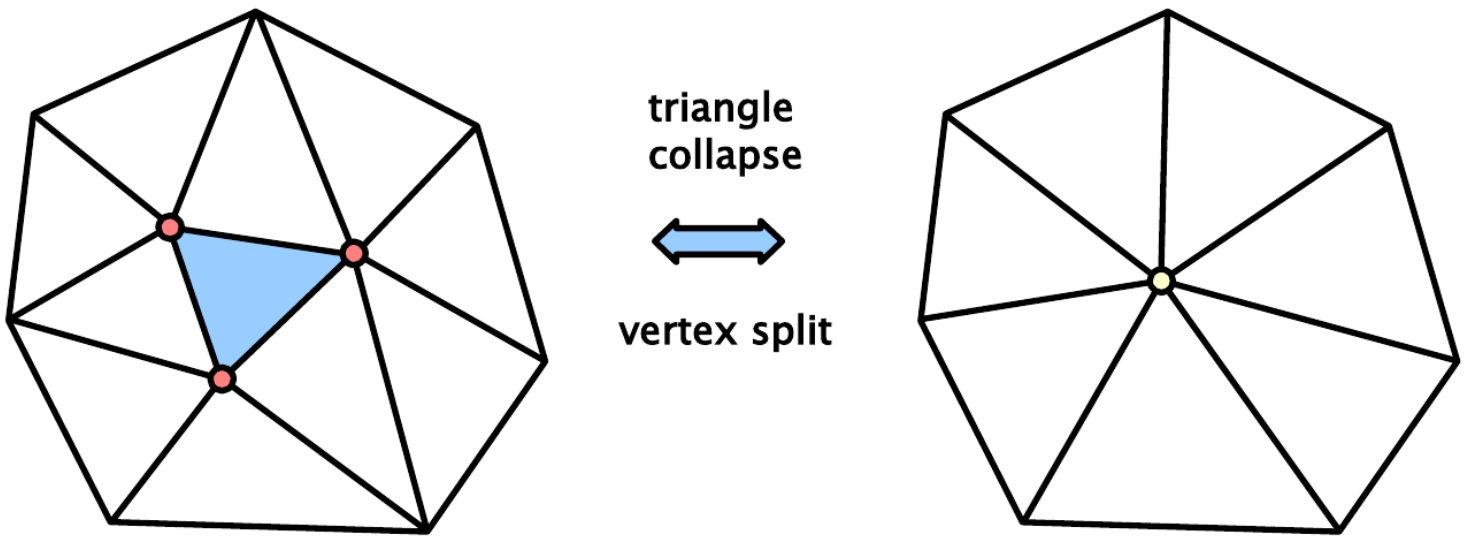
EDGE COLLAPSE VS. VERTEX SPLIT



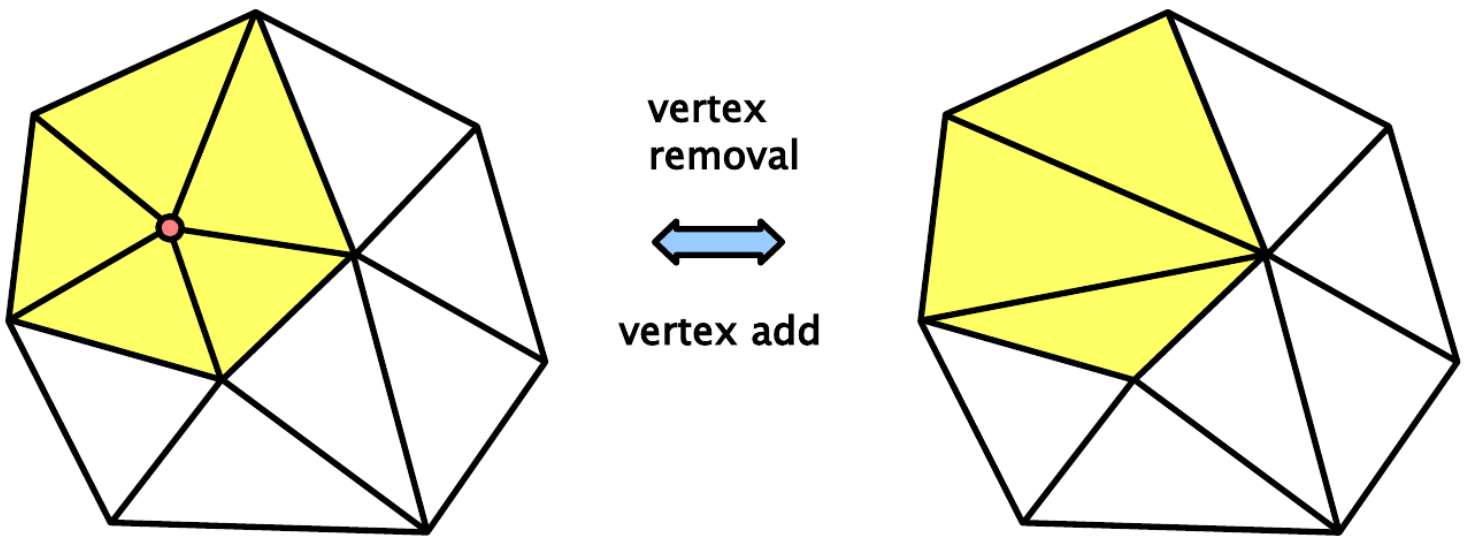
HALF-EDGE COLLAPSE VS. VERTEX SPLIT



TRIANGLE COLLAPSE COLLAPSE VS. VERTEX SPLIT



VERTEX REMOVAL VS. VERTEX ADD



TRIANGLE STRIPS/FANS

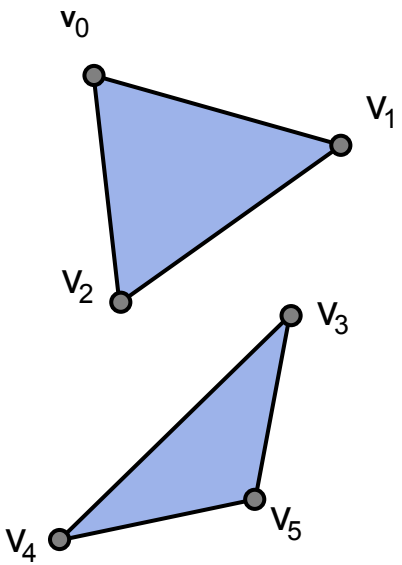


Figure:
GL_TRIANGLES

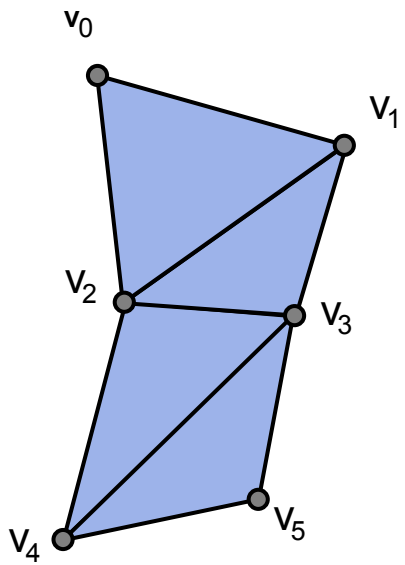


Figure:
GL_TRIANGLE_STRIP

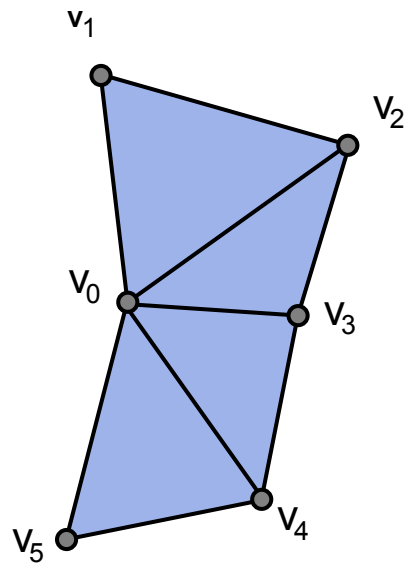


Figure:
GL_TRIANGLE_FAN

Limit Draw Calls

- Instanced rendering
- Multidraw commands – bundle more indexed render calls together

```
void glMultiDrawElements(GLenum mode, const GLsiz
```

- Indirect rendering
  - GL_DRAW_INDIRECT_BUFFER
  - Fill on GPU

```
void glDrawElementsIndirect()
void glDrawArraysIndirect()
void glMultiDrawElementsIndirect()
```

MULTITHREADING

- Contexts are not thread-safe
- Context can be current only in one thread
  - It can be used in multiple threads, but not in parallel
- Multiple contexts can be used concurrently
  - Enable resource sharing – textures, VBOs, ...
- Split preparation work between multiple threads (task systems)

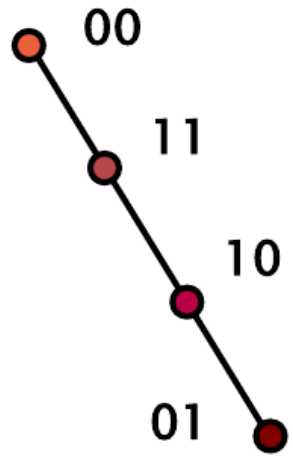
Textures

Texture Access

- Texture accesses expensive
  - Limit number of tex reads per fragment
- Use caching efficiently
  - Textures have spatial caching
  - Neighboring fragments should access neighboring texels
  - Use reasonable sized textures
  - Use mip-mapping

Texture Compression I

- Invention: S3 in DirectX 6 (1998)
  - DXTC, in OpenGL: S3TC, DXT1
- Fixed compression ratio
  - necessary for memory management
  - 4:1 to 6:1 lossy compression
- Decomposition into rectangle tiles (4×4 px)
  - each tile: two 16-bit colors and sixteen 2-bit indices (together – 4 bpp)
    - two extreme colors (R5G6B5), two more in-between colors (or one in-between and black)
    - each pixel is represented by a reference to one color
  - Extensions add alpha compression (128-bit): DXT3, DXT5



Texture Compression II

- NVIDIA VTC (Volume Texture Compression)
  - 3D variant of S3TC
  - data blocks $4 \times 4 \times 1$, $4 \times 4 \times 2$, $4 \times 4 \times 4$
- BPTC Texture Compression:
  - Byte stream
  - Unsigned + float
  - Multiple gradients

Other APIs

OpenGL ES

- OpenGL for Embedded Systems (smartphones, tablet computers, video game consoles)
- Most widely deployed 3D graphics API in history
- Subset of the OpenGL API
- Latest version of OpenGL ES 3.2

Evolution and Features of OpenGL ES Versions

- OpenGL ES 1.x: Fixed-function pipeline
- OpenGL ES 2.0: Programmable shading pipeline
- OpenGL ES 3.0: Higher quality graphics and textures, multiple render targets, 3D textures, etc.
- OpenGL ES 3.1: Introduces compute shaders, indirect draw commands, and enhanced texture functionality.
- OpenGL ES 3.2: Adds support for tessellation and geometry shaders

Differences in GLSL for OpenGL ES

- GLSL ES (OpenGL Shading Language for Embedded Systems) differs slightly from standard GLSL
- Key differences include:
  - Precision qualifiers (highp, mediump, lowp)
  - Some built-in functions and variables missing/modified
- The syntactic structure remains largely similar
- Versions:
  - OpenGL ES 2.0 – GLSL ES 1.00
  - OpenGL ES 3.x – GLSL ES 3.x0

Introduction to WebGL

- WebGL: a JavaScript API for rendering 3D graphics within any compatible web browser without the use of plug-ins.
- Based on OpenGL ES 2.0, allows the creation of GPU-accelerated graphics.
- Accessible through HTML5's $< canvas >$ element.

WebGL vs. OpenGL: Key Differences

- Runs in the web browser's JavaScript environment, offering easy integration with web content.
- Limited access to graphics hardware compared to full OpenGL for security and compatibility reasons.
- Automatically handles resource management to optimize for web performance and memory constraints.

Basic WebGL Setup

```
function initWebGL(canvas) {
    var gl = canvas.getContext('webgl');
    if (!gl) {
        alert('WebGL not supported');
        return;
    }
    return gl;
}
```

- Initialize WebGL context from a $< canvas >$ element.
- Check for browser support of WebGL.

Rendering a Triangle in WebGL

```
// Setup shaders, buffer data
var vertices = new Float32Array([
    0.0,  1.0,  // Vertex 1 (X, Y)
    -1.0, -1.0, // Vertex 2 (X, Y)
    1.0, -1.0  // Vertex 3 (X, Y)
]);
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// Draw the triangle
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

- Demonstrates setting up buffers and drawing a simple triangle.

WebGL 1 vs. WebGL 2

- **WebGL 1** – March 2011
  - Based on OpenGL ES 2.0.
  - Provides essential features for basic 3D graphics like texturing, shaders, and buffers.
  - Lacks transform feedback, 3D textures, multiple render targets, etc..
- **WebGL 2** – January 2017
  - Based on OpenGL ES 3.0.
  - Transform feedback, 3D textures, instanced rendering, and multiple render targets.
  - Shader precision, flexibility non-power-of-two textures and additional image formats.
- **Impact on Development**:
  - WebGL 1 – foundation for web-based 3D graphics, compatible across many devices.
  - WebGL 2 – more complex and visually appealing graphics.

Accelerated Graphics in Browsers

- **WebAssembly**: Enables high-performance applications on the web. Allows code written in languages like C/C++ to be compiled to run in the browser near-native speed, enhancing complex graphics and compute tasks.
- **WebGPU**: An upcoming standard intended to provide modern 3D graphics and computation capabilities. It exposes GPU hardware power similar to Vulkan, Direct3D 12, and Metal, aiming to supersede WebGL with better efficiency and control.

DirectX (Direct3D) I

- Available only for Microsoft platforms (Windows, XBox)
- Wraps same hardware – similar principles to OpenGL
- API organization different
  - State machine vs. parameter objects
  - Resources managed by developer
- Component Object Model (COM)
  - MS binary-interface standard for software components
  - Support for interprocess communication, multiple languages
  - Entities (textures, render targets, shaders, . . .) are COM objects
- Shading language HLSL
  - Similar to GLSL (almost 1:1 mapping)
  - Cross compilers available

DirectX (Direct3D) II

- Not a standard – more flexible to introduce changes (backward incompatible)
- DirectX 9 $\longleftrightarrow$ OpenGL 2
- DirectX 10 $\longleftrightarrow$ OpenGL 3
- DirectX 11 $\longleftrightarrow$ OpenGL 4
- DirectX 12 $\longleftrightarrow$ Vulkan, Metal

DIRECT 3D

Basic interfaces:

```
IDXGISwapChain *swapchain;              // the pointer to the swap chain interface
ID3D11Device *dev;                      // the pointer to our Direct3D device interface
ID3D11DeviceContext *devcon;            // the pointer to our Direct3D device context
```

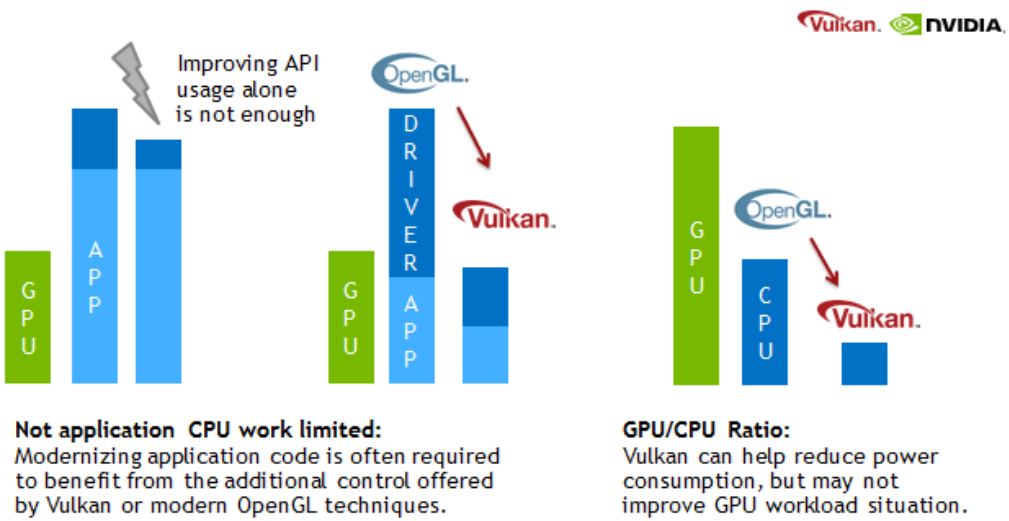
- ID3D11Device thread-safe
  - Can prepare resources in multiple threads
- ID3D11DeviceContext must use some synchronization
  - Draw commands cannot be issued in parallel

New APIs

- Opengl, DirectX (11 and older)
  - Old APIs – designed fo different ecosystems
  - Hotfixed for demands of modern hardware and software architectures
- CPU time in driver – hot spot in lots of modern applications
  - Multicore CPUs cannot efficiently feed command to GPUs
- AMD Mantle $\rightarrow$ Vulkan, Metal (Apple)
  - Explicit command buffer control
  - Multithreaded parallel rendering
  - Lower level API – developers have more control (also more code is needed)
- DirectX 12
  - Added similar features
  - Command queues, feeding from multiple threads

VULKAN

- Managed by Khronos
- Same API for desktop and embedded systems
- Low CPU usage, efficient usage of multi-core systems (thread-safe)
- No global state
- GLSL compiled to SPIR-V (Standard Portable Intermediate Representation)
- OpenCL convergence to Vulkan



Command Buffers and Command Pools

- Commands are recorded in command buffers before submission to GPU.
- Efficient reuse and better multithreading capabilities.
- Command pools manage the memory for command buffers.

Queue Families and Synchronization

- Queues are used for submitting commands; each has a specific function.
- Explicit synchronization is required to manage resource access.
- Ensures that resources are not read and written at the same time.

Pipeline State Objects (PSOs)

- All state settings required for a draw call are compiled into a PSO.
- Changes in state settings require switching PSOs.
- Leads to more optimized state handling.

VULCAN USAGE I

- Create a VkInstance
  - Specify required features, extensions
- Select graphics card (VkPhysicalDevice)
  - Listing cards supporting requested features
- Create a logical device (VkDevice)
- Create queues (VkQueue) from queue families
  - Graphics
  - Compute
  - Memory transfers
- Create a window, window surface and swap chain

VULCAN USAGE II

- Wrap the swap chain images into VkImageView
- Create a render pass
  - Specifies the render targets and usage
- Create framebuffers for the render pass
- Set up the graphics pipeline (VkPipeline)
  - Configured in advance
- Allocate and record a command buffer with the draw commands
- Draw frames
  - Acquire images
  - Submit draw commands
  - Return images to swap chain
  - Synchronization required