Realtime Computer Graphics on GPUs

Exam Notes by Jakub Kos

Disclaimer: These notes are based on the lecture series by *Jan Kolomaznik*. They are intended as a concise summary for revision purposes. For a complete understanding, I highly recommend reviewing all original lecture slides and materials.

COURSE BRIEF

- GPU Architecture, History
  - simplified programmable pipeline (vertex, fragment shaders)
  - basic primitives, rasterisation, 2D rendering
  - historical background - 3D rendering - wireframe, flat, basic lighting, Gourand vs. Phong shading
  - Phong model
- Math
  - 2D linear transformation, rotation by formula, matrix, complex numbers
  - 3D transformations - affine, perspective space
  - 3D rotations - euler angles, gimbal lock, matrices
  - Normal matrices
  - Quaternions
  - lerp, slerp
  - easing/tweening
  - animation curves
- Textures
  - coordinates
  - aliasing vs. filtering
  - mip-maps
  - multitexturing
  - bump mapping
  - 3D textures
- Curves and Surfaces
- Framebuffer
  - render to texture
  - deferred shading
  - antialiasing
  - stencil buffer - shadow-map, shadow volume, mirrors
  - effects in screen space (ambient occlusion, DOF, . . .)
- Generating geometry
  - datastructures
  - tesselation shaders
  - geometry shaders
  - mesh shaders
- Speedup techniques
  - near/far clipping
  - occlusion culling
  - instancing
  - billboards, decals
  - LOD
  - triangle fan, strip
- Advanced techniques, Effects
  - bindless textures
  - megatextures
  - volume rendering
  - CAD visualization
  - scientific visualization
- Other technologies
  - OGL ES
  - WebGL
  - Vulkan
  - DX11, DX12
  - Optix + raytracing
- GPGPU
  - compute shaders
  - OpenCL
  - CUDA
    - terminology
    - OGL interoperability
    - computation model
    - memory types
  - Deep learning

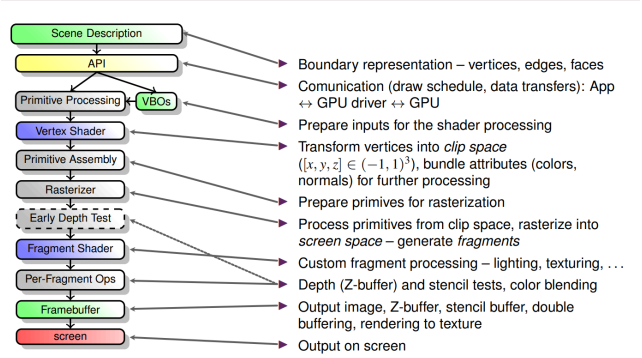
- Realtime Raytracing
- CSG, Depth peeling, Trasform feedback
- Animation

Introduction

Realtime algorithms

- Time Constrains:
  - Hard limit
  - Soft limit
- Optimal algorithm (time complexity ?)
- Approximations vs. precision requirements
- Tuning for specific hardware
- Specialized tools for hot spots - **GPUs**

Programmable Pipeline



OpenGL

- Open standard
  - OpenGL Architecture Review Board (ARB) 1992-2006
  - Khronos Group 2006
- Current version 4.6 (released in 2017)
- Multiplatform, language-independent
- Additional functionality possible by HW vendor extensions
- Open source implementation - Mesa
  - Acting as both a driver and translation layer to other graphics APIs.
- Concepts
  - API formed by set of functions and integer constants
  - Asynchronnous calls (queries) for efficient CPU/GPU parallelism
  - GL context - internal global state owning OGL objects
    - Multiple contexts possible (data sharing, etc.)
    - API calls make modification to current context
  - OGL objects (textures, buffers, framebuffers, shader programs, . . .)
    - Gen/Delete paradigm - glGen*(GLsizei n, GLuint* objects), glDelete(GLsizei n, const GLuint *objects)
    - Bind before usage - glBind*(GLenum target, GLuint object)

```
float vertices [] = {
-0.5f , -0.5f ,
0.5f , -0.5f ,
0.0f , 0.5f ,
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO) ;
// bind the Vertex Array Object first,
// then bind and set vertex buffer(s), and then configure vertex attribute(s).
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// index, size, type, normalized, stride, pointeroffset
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2*sizeof(float), (void*) 0);
glEnableVertexAttribArray(0);
```

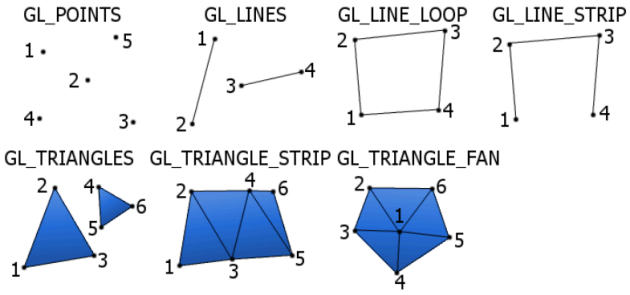
- VBO is basically an array of data thats stored for every vertex.
- EBO is an array of data thats describes how every element (aka triangle) is constructed.
- VAOs bundle together all these buffers (like a container of pointers) and allow you to utilize them easier.

Draw calls

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 3)
glDrawArrays(GL_LINE_LOOP, 0, 3);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Rasterizer

- Decomposition of vector primitives into fragments
- Fragment:
  - Raster element - potentialy attributes to pixel color
  - Size: same or smaller (antialiasing) then the target pixel
- Interpolation of vertex attributes - linear, barycentric coordinates
- Triangles sharing two vertices - no overlap, no gap



OpenGL shading language (GLSL)

- C based programming language
- Programmable pipeline customization

```
// vertex shader
int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
// fragment shader
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
// link shaders
int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Vertex shader

```
#version 430

in vec3 in_vert;

void main() {
    gl_Position = vec4(in_vert, 1.0);
}
```

Fragment shader

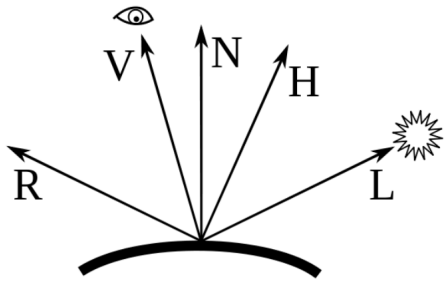
```
#version 430
// Same color for all fragments and all draw calls (Use same color for all fragments)
const vec3 color = vec3(1.0, 0.0, 0.0);
// Same color for all fragments per draw call + default <- value (Customize color from CPU side)
uniform vec3 color = vec3(1.0, 0.0, 0.0);

out vec4 out_color;
void main() {
    out_color = vec4(color, 1.0);
}
```

Obligations (VS <-> FS)

- VS obligation: vertex coordinates in “clip space”
  - for 3D primitive rasterizing
  - other output varying data are optional (texture coordinates, normals, primary and secondary color, etc.)
- VS-FS cooperation
  - GPU is not aware of data semantics:
  - rasterizer unit usually interpolates all the data (perspective correct interpolation)
  - flat option (prevents the interpolation)

OpenGL FFP - Lighting Model

- Gouraud shading
- Blinn-Phong reflection model
- Hardcoded maximum number of available lights



$$\text{Phong:}\quad I_p = k_a\, i_a + \sum_{m \in \text{lights}} \Big(k_d\, (L_m \cdot N)\, i_{m,d} + k_s\, (R_m \cdot V)^\alpha\, i_{m,s} \Big),$$
$$R_m = 2\, (L_m \cdot N)\, N - L_m$$

$$\text{Blinn-Phong:}\quad I_p = k_a\, i_a + \sum_{m \in \text{lights}} \Big(k_d\, (L_m \cdot N)\, i_{m,d} + k_s\, (N \cdot H)^\alpha\, i_{m,s} \Big),$$
$$H = \text{normalize}(L + V)$$

The code?

OPENGL CALLS

GLSL DATA TYPES

Math

Vector Operations

Scalar (Dot) Product

Definition:
$\mathbf{p} \cdot \mathbf{q} \;=\; \sum_{i=0}^{n-1} p_i\, q_i$

Value:
$\mathbf{p} \cdot \mathbf{q} \;=\; \|\mathbf{p}\|\, \|\mathbf{q}\|\, \cos\alpha$

Matrix notation:

$$\mathbf{p} \cdot \mathbf{q} = \mathbf{p}^\mathsf{T}\, \mathbf{q} = \begin{bmatrix} p_0 & p_1 & \cdots & p_{n-1} \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ \vdots \\ q_{n-1} \end{bmatrix}$$

Vector Projection

Projection onto another vector:

$$\mathbf{p}_{\text{proj}} = \frac{\mathbf{p} \cdot \mathbf{q}}{\|\mathbf{q}\|} \mathbf{q}$$

Matrix notation $(\mathbf{q}\,\mathbf{q}^T)$:

$$\mathbf{p}_{\text{proj}} = \frac{1}{\|\mathbf{q}\|^2} \begin{bmatrix} q_x^2 & q_x\,q_y & q_x\,q_z \\ q_x\,q_y & q_y^2 & q_y\,q_z \\ q_x\,q_z & q_y\,q_z & q_z^2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Cross Product

Definition:

$$\mathbf{p} \times \mathbf{q} = \big[\, p_y q_z - p_z q_y,\; p_z q_x - p_x q_z,\; p_x q_y - p_y q_x \big]$$

As formal determinant:

$$\mathbf{p} \times \mathbf{q} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ p_x & p_y & p_z \\ q_x & q_y & q_z \end{vmatrix}$$

Matrix formulation:

$$\mathbf{p} \times \mathbf{q} = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

Perpendicularity

$$(\mathbf{p} \times \mathbf{q}) \cdot \mathbf{p} \;=\; (\mathbf{p} \times \mathbf{q}) \cdot \mathbf{q} \;=\; 0$$

Magnitude

$$\big\|\mathbf{p} \times \mathbf{q}\big\| \;=\; \big\|\mathbf{p}\big\|\,\big\|\mathbf{q}\big\|\,\sin\alpha$$

Orientation
Follows the right-hand rule.

Rotations

2D Rotation

Basic expression:

$$x' = x\cos\alpha \;-\; y\sin\alpha, \qquad y' = x\sin\alpha \;+\; y\cos\alpha$$

Matrix notation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Complex exponential:

$$[x, y] \implies z = x + i\,y$$

- Rotate by multiplying z by $e^{i\alpha} = \cos\alpha + i\sin\alpha$
- Inverse rotation via the complex conjugate of $e^{i\alpha}$

Elementary Rotations in 3D

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation Around an Arbitrary Axis

Assume Axis a, angle θ, point p, rotated point p' and $\|\mathbf{a}\| = 1$.

- **Project $\mathbf{p}$ onto the axis**

$$\mathbf{p}_{\text{proj}} = (\mathbf{a} \cdot \mathbf{p})\,\mathbf{a}, \qquad R_{\mathbf{a},\theta}\,\mathbf{p}_{\text{proj}} = \mathbf{p}_{\text{proj}}$$

- **Perpendicular component**

$$\mathbf{p}_{\perp} = \mathbf{p} \;-\; (\mathbf{a} \cdot \mathbf{p})\,\mathbf{a}, \qquad \|\mathbf{p}_{\perp}\| = \|\mathbf{p}\| \sin\alpha$$

- **Cross product with the axis**

$$(\mathbf{a} \times \mathbf{p}) \cdot \mathbf{p}_{\perp} = 0, \qquad \|\mathbf{a} \times \mathbf{p}\| = \|\mathbf{p}\| \sin\alpha$$

- **Final rotated position:**

$$\mathbf{p}'_{\perp} = \mathbf{p}_{\perp} \cos\theta \;+\; (\mathbf{a} \times \mathbf{p})\, \sin\theta, \qquad \mathbf{p}' = \mathbf{p}'_{\perp} + \mathbf{p}_{\text{proj}}$$

Matrix representation:

$$\mathbf{p}'_{\perp} = \big[\mathbf{p} - (\mathbf{a} \cdot \mathbf{p})\,\mathbf{a}\big] \cos\theta + (\mathbf{a} \times \mathbf{p})\, \sin\theta = \mathbf{p}\cos\theta + (\mathbf{a} \times \mathbf{p}) \sin\theta + \mathbf{a}\,(\mathbf{a} \cdot \mathbf{p})(1 - \cos\theta)$$

$$\mathbf{p}' = \Big(I\cos\theta \;+\; [\mathbf{a}]_{\times}\, \sin\theta \;+\; \mathbf{a}\,\mathbf{a}^{T}\,(1 - \cos\theta)\Big)\,\mathbf{p}$$

where

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad [\mathbf{a}]_{\times} = \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix}, \quad \mathbf{a}\,\mathbf{a}^{T} = \begin{bmatrix} A_x^2 & A_x A_y & A_x A_z \\ A_x A_y & A_y^2 & A_y A_z \\ A_x A_z & A_y A_z & A_z^2 \end{bmatrix}.$$

Final matrix form:
Let $c = \cos\theta$, $s = \sin\theta$, and $\mathbf{A} = (A_x, A_y, A_z)$ be a unit axis. Then the rotation matrix is

$$R_{\mathbf{a},\theta} = \begin{bmatrix} c + (1-c)A_x^2 & (1-c)A_x A_y \;-\; s\,A_z & (1-c)A_x A_z \;+\; s\,A_y \\ (1-c)A_x A_y \;+\; s\,A_z & c + (1-c)A_y^2 & (1-c)A_y A_z \;-\; s\,A_x \\ (1-c)A_x A_z \;-\; s\,A_y & (1-c)A_y A_z \;+\; s\,A_x & c + (1-c)A_z^2 \end{bmatrix}$$

Euler angles

- arbitrary rotation decomposed into three components
- Leonard Euler (1707-1783)
- 3 angles - 3 elementary rotations
- order of rotations important (x-y-z, roll-pitch-yaw, z-x-z, …)
  - intrinsic vs. extrinsics
- Disadvantages:
  - Problematic interpolation between two orientations
  - Gimbal lock - not as severe in SW as in HW (Apollo)

Quaternions

- generalization of complex numbers in 4D space
- usage in graphics since 1985 (Shoemake)
- $q = (v, w) = ix + jy + kz + w = v + w$
- imaginary part $v = (x, y, z) = ix + jy + kz$
- $i^2 = j^2 = k^2 = -1, \quad jk = -kj = i, \quad ki = -ik = j, \quad ij = -ji = k$

It's better to watch some video to understand, for example Quaternions and 3d rotation, explained interactively.

Summary

- **rotational matrix**
  - $+$ HW support, efficient point/vector transformation
  - $-$ memory (float[9]), other operations are not so efficient
- **rotational axis and angle**
  - $+$ memory (float[4] or float[6]), similar to quaternion

    - $-$ inefficient composition and interpolation
- **quaternion**
    - $+$ memory (float[4]), composition, interpolation
    - $-$ inefficient point/vector transformation

Affine and projective spaces

Affine space: - Set V of vectors and set P of points - Affine transformations can be represented by matrix

Projective space: - Homogeneous coordinates - All lines intersect (space contains infinity) - Affine and projective transformations can be represented by matrix

Homogenneous coordinates

- homogeneous coordinate vector $[x, y, z, w]$
- transformation: *multiplying by a* 4×4 *matrix*
- homogeneous matrix is able to translate and to do perspective projections
- from homogeneous coordinates $[x, y, z, w]$ into Cartesian coordinates:
    - by division $(w \neq 0)[x/w, y/w, z/w]$
- coordinate vector $[x, y, z, 0]-$ point in infinity
- from Cartesian coordinates to homogeneous:
    - trivial extension $[x, y, z]...[x, y, z, 1]$

Transformation Matrix

$$\mathbf{A}\,\mathbf{p} = \begin{bmatrix} \mathbf{M} & \mathbf{T} \\ 0\;0\;0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix}$$

- $\mathbf{T}$ defines translation
- $\mathbf{M}$ defines:
    - rotation:

      $$\mathbf{M}_{\text{rot}} = R_z(\gamma)\; R_y(\beta)\; R_x(\alpha),$$

      where

      $$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, \quad R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}, \quad R_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

      together

      $$\mathbf{M}_{\text{rot}} = \begin{bmatrix} \cos\beta\cos\gamma & -\cos\beta\sin\gamma & \sin\beta \\ \cos\alpha\sin\gamma + \sin\alpha\sin\beta\cos\gamma & \cos\alpha\cos\gamma - \sin\alpha\sin\beta\sin\gamma & -\sin\alpha\cos\beta \\ \sin\alpha\sin\gamma - \cos\alpha\sin\beta\cos\gamma & \sin\alpha\cos\gamma + \cos\alpha\sin\beta\sin\gamma & \cos\alpha\cos\beta \end{bmatrix},$$

      > I would recommend praying to god that the matrix will be Identity
    - scaling:

      $$\mathbf{M}_{\text{scale}} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$
    - shear:

      $$\mathbf{M}_{\text{shear}} = \begin{bmatrix} 1 & 0 & \lambda \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- (and any combination of the above)

Normal Vector Transformation

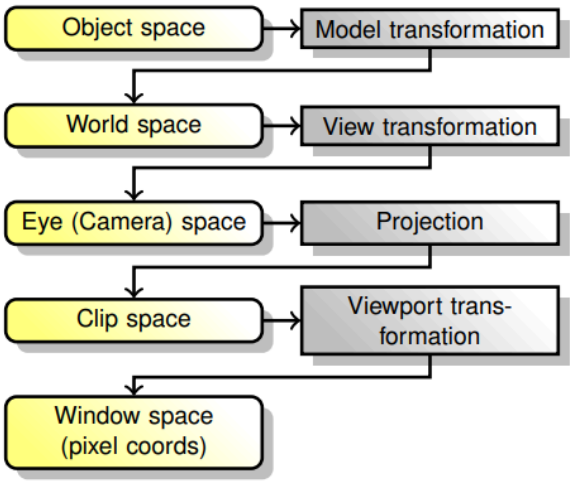
- Only orientation change is valid transformation for normals
- Tangents ($\mathbf{t}$) remain valid:

$$\mathbf{n} \cdot \mathbf{t} = 0 \implies \mathbf{n}' \cdot \mathbf{t}' = (G\,\mathbf{n}) \cdot (M\,\mathbf{t}) = 0$$

$$(G\,\mathbf{n}) \cdot (M\,\mathbf{t}) = (G\,\mathbf{n})^T\,(M\,\mathbf{t}) = \mathbf{n}^T G^T M\,\mathbf{t} \implies G = (M^{-1})^T$$

Transforma

Transformations for rendering pipeline



LookAt Camera Matrix

Given camera **eye** position $\mathbf{e}$, look-at point $\mathbf{p}$, and up vector $\mathbf{u}$:

$$\mathbf{v} = \text{norm}(\mathbf{e} - \mathbf{p}), \qquad \mathbf{n} = \text{norm}(\mathbf{v} \times \mathbf{u})$$

The 4×4 transform that moves the camera to its position and orientation is

$$\text{TR} = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x & u_x & v_x & 0 \\ n_y & u_y & v_y & 0 \\ n_z & u_z & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

World view needs to be transformed by its inverse:

$$(\text{TR})^{-1} = R^{-1}\,T^{-1} = R^T\,T^{-1} = \begin{bmatrix} n_x & n_y & n_z & 0 \\ u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which multiplies out to:

$$(\text{TR})^{-1} = \begin{bmatrix} n_x & n_y & n_z & -(\mathbf{n} \cdot \mathbf{e}) \\ u_x & u_y & u_z & -(\mathbf{u} \cdot \mathbf{e}) \\ v_x & v_y & v_z & -(\mathbf{v} \cdot \mathbf{e}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection (Frustum)

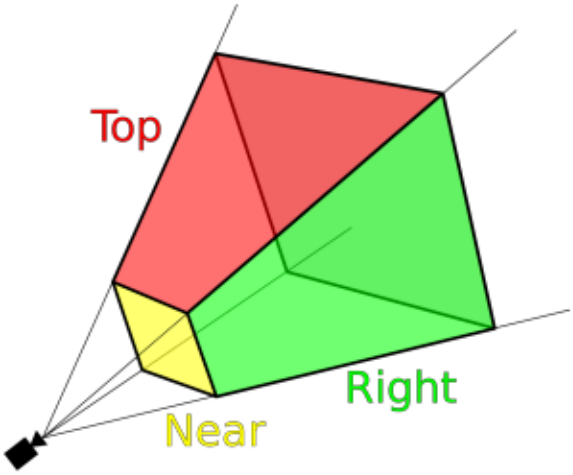
Point $\mathbf{p} = (p_x, p_y, p_z)$ projection:

$$x = -\frac{n}{p_z}\,p_x, \quad y = -\frac{n}{p_z}\,p_y$$

Frustum projection matrix:

$$P_{\text{frustum}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2\,f\,n}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note: Perspective-correct interpolation requires dividing by the clip-space w (here $w = -p_z$) after this transform.



Textures

- Appearance enhancement:
  - color modulation with raster images (“bitmap”)
  - bump-mapping to fake geometric detail
  - possible modulation of transparency, reflectance, environment light
- Texture definition:
  - 1D or 2D data array (“bitmap texture”)
    - widely used, broad hardware support
  - 3D data array (“volume texture”)
  - procedural - callback algorithm in every fragment (programmable GPU)

Texture access

Texturing API

- Texture handle creation: `cpp unsigned int texture; glGenTextures(1, &texture)`
- Texturing unit activation and texture binding: `cpp glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, texture);`
- Data upload: `cpp glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);`
- Texturing parameters: `cpp glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); ...`

Texture mapping

- 2D textures have to be mapped to an object surface
  - texture coordinates $[u, v]$ ($[s, t]$ in OpenGL) defined in every vertex
  - GPU interpolates them correctly into individual fragments
  - bitmap data need to be interpolated (among adjacent texture pixels = “texels”)

Texture unwrap

- Cut along *seam edges*
- Flatten geometry to minimize distortion (prevent stretched faces)

Texture repetition

- Standard texture-coordinates domain: $[0, 1]^D$
  - Handle out-of-range values?
- Cyclic repetition (`GL_REPEAT`)
- Mirroring (`GL_MIRRORED_REPEAT`)
  - Every other tile is flipped -> better continuity
- Clamping (`GL_CLAMP`, `GL_CLAMP_TO_EDGE`)
  - Optional explicit border color (`GL_CLAMP_TO_BORDER`)
  - Can be used for debugging (special color)

Texture combination

- Modern GPUs (since TNT) can combine more textures in one fragment ("multitexturing")
  - global (low-frequency) basis + detail texture
  - pre-computed lighting ("light-map")
  - "environment maps" - reflection of a surround scene
- Legacy combination operators:
  - `REPLACE` (source is ignored)
  - `MODULATE` (multiplication - values are abated)
  - `DECAL` (semi-transparent texture on an original surface)
  - `INTERPOLATE` (lerp, 2 sources)
  - `DOT3_RGB[A]` (inner product, for 3D)
  - `ADD`, `ADD_SIGNED`, `SUBTRACT`, …
- programmable GPU (in "fragment shader"): arbitrary formula

Texture mapping units

- Hardware component for processing texels
- One texture mapping unit (TMU) handles one bitmap source
- Two jobs:
  - **Texture Addressing**: texture coordinates -> texels -> fragments (pixels)
  - **Texture Filtering**: interpolation, filtering
- Modern hardware - multiple texture units (one texture processed by multiple HW units)
- More TMUs -> higher fill rate
- Spatial caching - neighboring fragments access texel from small neighborhood

Samplers

- Sampling parameters for a texture access inside of a shader
- `glBindSampler()` + `glBindTexture()` - bind to a texture unit

```
# version 330

in vec2 v_tex;
out vec4 f_color;

uniform sampler2D u_texture;

void main () {
    f_color = texture(u_texture, v_tex);
}
```

Examples

- Most frequently used approaches:
  - gloss mapping (glossy reflection)
  - light mapping (alt: dark mapping) - lighting, shadow mapping - pre-computed shadow, ambient occlusion
    - Precompute lighting effects
    - Bake into light/shadow map
    - Static lighting - light source cannot be moved
  - bump mapping (normal-vector modulation)
    - special texturing technique - impression of a bumpy surface
      - replaces complicated macro-geometry
      - modifies (modulates) normal vector in every pixel
      - Phong shading (normal interpolation) is recommended
      - human observer thinks that a surface is actually bumpy (much of the impression is inferred from specular reflections)

- parallax mapping (texture coordinates modulation)
  - simulate parallax
    - modulate texture coordinates based on displacement map
    - used together with bump mapping
- environment mapping (environment reflection)
  - reflection vector R converted to
    - spherical coordinates - more complicated
    - six cube faces - “cube mapping”

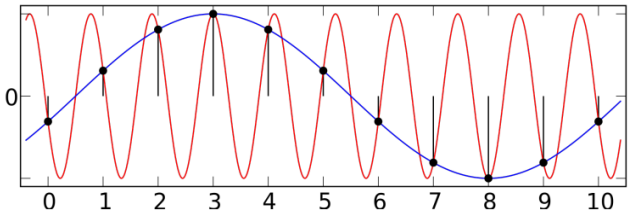
Filtering

Aliasing

- Reconstruction of original signal from discrete samples
- Problem when sampling frequency (f_{sampl}) below Nyquist limit:

$$f_{sampl} < 2f_{max}$$

- Shannon theorem
- Aliasing examples and preventions:
  - Moire pattern (interference), rasterization
  - high speed rotation + camera, rolling shutter
  - fluorescent light + lathe
  - CD-quality audio sampling frequency



Aliasing prevention

- Higher sampling frequency
- Preprocess signal - correctly remove high frequencies (low-pass filtering)
- Hide artefacts behind another (less disturbing phenomenon) - random noise

Texture filtering

- texture “seen from a distance” should be filtered (raster image sub-sampling)
  - otherwise “alias” will appear (especially disturbing in motion)
- pre-processing techniques
  - MIP-map (“multum in parvo”), most popular (HW)
  - RIP-map, anisotropic miniatures
  - anisotropic filtering - dynamic method, MIP-map + number of linear samples
  - summary tables - pre-computed upper-left rectangle sums

MIP-Mapping

- texture subsampling in advance - binary fractional resolutions (1/4, 1/16, etc. - HW supported)
  - high quality sub-sampling with averaging
  - 3-component color (RGB) - convenient arrangement in memory
  - `glGenerateMipmap()`
- MIP-map utilization
  - compute level (according to required texture scaling)
  - either single texel fetch (speed)
  - or interpolation between two adjacent MIP-map levels or even bi-linear interpolation in the levels (at most 8 fetches = quality)

Anisotropic filtering

- back-projected screen pixel = deformed quadrangle
- MIP-map level according the higher sub-sampling (shorter size)
- multi-sampling (averaging) along the longer side

Custom filtering

- Arbitrary filtering implemented in shader
- Integral images (summary tables)
- Multiple texture accesses
  - Incorporate perspective (anisotropy):
  - Derivatives between fragments: dFdx(), dFdy():

Example: flat normal

```
normalize(cross(dFdx(pos), dFdy(pos)));
```

3D Textures

- Trilinear interpolation
- Modeling material properties (marble, wood, clouds)
- Z-direction interpreted as time - animation
- Precomputed lighting effects: normal -> texture coordinates
- Scientific applications
  - Tomography
  - Vector fields - fluid simulations,…
- Application (Medical data visualization)
  - Maximum intensity projection
  - Density integration
  - Isosurfaces
  - 1D transfer function

Curves and Surfaces

Curve and surface definition

Explicit:

$$y = \sin(x), \qquad z = x^2 + y^2$$

Implicit:

$$0 = x^2 + y, \qquad 0 = x^2 + y^2 + z^2$$

Parametric:

$$\mathbf{Q}(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix}, \qquad \mathbf{P}(u,v) = \begin{bmatrix} \cos(u)\left(R + r\cos(v)\right) \\ \sin(u)\left(R + r\cos(v)\right) \\ r\ \sin(v) \end{bmatrix}$$

Parametric curves and surfaces

- Parameter(s) have specified range ($[0,1]$)
- Same curve (surface)can have multiple parametrizations
  - Arc length parametrization - animation, uniform sampling,…
  - In general case cannot be expressed analyticaly
- Sampling the parameter space -> discrete points in space
  - Approximating by polyline or polygonal mesh

Cubic curves

- Basic parametric cubic curve equation:

$$\mathbf{Q(t)} = \mathbf{a} + \mathbf{b}\,\mathbf{t} + \mathbf{c}\,\mathbf{t^2} + \mathbf{d}\,\mathbf{t^3}$$

$$\begin{aligned} Q_x(t) &= a_x + b_x\,t + c_x\,t^2 + d_x\,t^3, \\ Q_y(t) &= a_y + b_y\,t + c_y\,t^2 + d_y\,t^3, \\ Q_z(t) &= a_z + b_z\,t + c_z\,t^2 + d_z\,t^3. \end{aligned}$$

Rewrite in matrix form:

$$\mathbf{Q}(t) = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

Compact notation:

$$\mathbf{Q}(t) = C\,T(t)$$

where

$$C = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix}, \quad T(t) = \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}.$$

Simple derivative:

$$\mathbf{Q}'(t) = C\,\frac{d}{dt}T(t) = C\begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \end{bmatrix}$$

Geometrical constraints

- Certain curves can be defined as weighted sum of four geometrical constraints.
- B_k - blending functions:

$$\begin{aligned} \mathbf{Q}(t) &= \sum_{k=0}^{3} B_k(t)\,\mathbf{g}_k \\ &= (a_1 + b_1 t + c_1 t^2 + d_1 t^3)\,\mathbf{g}_1 \\ &+ (a_2 + b_2 t + c_2 t^2 + d_2 t^3)\,\mathbf{g}_2 \\ &+ (a_3 + b_3 t + c_3 t^2 + d_3 t^3)\,\mathbf{g}_3 \\ &+ (a_4 + b_4 t + c_4 t^2 + d_4 t^3)\,\mathbf{g}_4 \end{aligned}$$

- Rewrite in a matrix form:

$$\mathbf{Q}(t) = \begin{bmatrix} \mathbf{g}_1 & \mathbf{g}_2 & \mathbf{g}_3 & \mathbf{g}_4 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

Compact matrix form:

$$\mathbf{Q}(t) = G\,M\,T(t)$$

where $\mathbf{G}$ geometry matrix, $\mathbf{M}$ basis matrix.

Hermite curves

- **Endpoints** $\mathbf{P}_1,\ \mathbf{P}_2$
- **Tangents** at those points $\mathbf{T}_1,\ \mathbf{T}_2$

$$G = \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix}$$

$$M_H = \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Catmull-Rom splines

- **Control points $\mathbf{P}_1, \ldots, \mathbf{P}_n$**
- **Tangent at point**

$$\mathbf{T}_i \;=\; \tfrac{1}{2}\left(\mathbf{P}_{i+1} - \mathbf{P}_{i-1}\right)$$

- **Catmull-Rom -> Hermite**

$$G_H \;=\; \begin{bmatrix}\mathbf{P}_{i-1} & \mathbf{P}_i & \mathbf{P}_{i+1} & \mathbf{P}_{i+2}\end{bmatrix}$$

$$M_{\text{CR}} = \tfrac{1}{2}\begin{bmatrix}0 & -1 & 2 & -1\\ 2 & 0 & -5 & 3\\ 0 & 1 & 4 & -3\\ 0 & 0 & -1 & 1\end{bmatrix}$$

Bézier Curves

- **P. de Casteljau (Citroën)** — numerically stable algorithm using linear interpolations
- **P. Bézier (Renault)** — Bernstein polynomials as blending functions

$$B_{n,k}(t) \;=\; \binom{n}{k} t^k \left(1-t\right)^{n-k}$$

- Four control points per cubic segment

$$\mathbf{B}(t) \;=\; \sum_{k=0}^{3} B_{3,k}(t)\,\mathbf{P}_k$$

- **Basis matrix**

$$G_B = \begin{bmatrix}\mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 & \mathbf{P}_4\end{bmatrix}$$

$$M_B = \begin{bmatrix}1 & -3 & 3 & -1\\ 0 & 3 & -6 & 3\\ 0 & 0 & 3 & -3\\ 0 & 0 & 0 & 1\end{bmatrix}$$

- **de Casteljau algorithm**
  - Repeated linear interpolation
  - Each lerp uses same weight

Surfaces

Bicubic surfaces

- Extension of bilinear interpolation concept
- Tensor product surfaces
- 16 geometrical constraints - $\mathbf{G} - 4 \times 4 \times 3$ coefficients

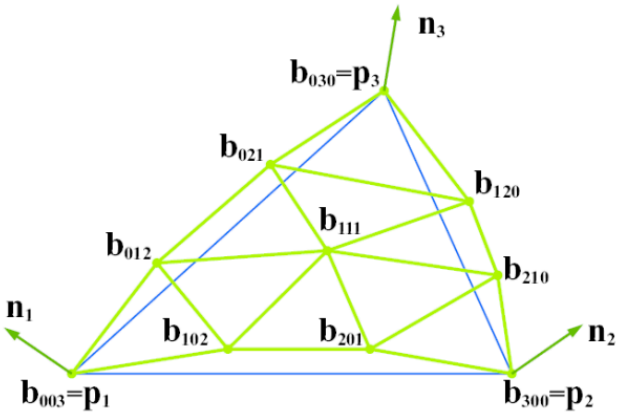
Tensor-product form:

$$\mathbf{Q}(u,v) = \sum_{k=0}^{3}\sum_{l=0}^{3} B_k(u)\,B_l(v)\,\mathbf{g}_{kl}$$

Matrix form:

$$\mathbf{Q}^r(u,v) = \mathbf{S}^T(v)\;M^t\;G^r\;M\;T(u)$$

- **Bezier triangle**
- de Casteljau extension for triangles - barycentric coordinates instead of lerp
- 10 control points
  - Simplification for triangle meshes with vertex normals - PN-triangles



Subdivision surfaces

- Recursive algorithms
- Two steps:
  - Refine mesh topology
  - Adjust vertex positions
- Interpolating vs approximating

Catmull-Clark scheme

- Produces quad mesh - keeps clean topology
- Vertices inserted into edges and face centers

Loop scheme

- Defined for triangle meshes
- Spliting edges
- New position - weighted average of vertices from incident triangles

---

Framebuffer and Offscreen Rendering Techniques

Double buffering

- Single frame buffer problems:
  - screen tearing
  - flickering
  - render artefacts
- Double buffering - also known as *page flipping*
  - **Front buffer** - currently visible
  - **Back buffer** - currently rendered off-screen
- Requires fast buffer swap

Framebuffer Structure

- Default framebuffer created with window creation
- Custom off-screen framebuffer:
  - Can choose resolution
  - Arbitrary attachments
  - Render to texture
  - Filtering, postprocessing
  - Interoperability with other APIs (`CUDA`, `OpenCL`, …)

Framebuffer attachments

- 2D rendering target
- Almost any object containing image or image array
- For complex objects specify what part to attach:
  - Cube map select face
  - 3D texture z-slice
  - Mipmap choose a level…
- Specify semantics - how it will be used in the rendering pipeline
- **Color attachments**

    - Should match fragment shader outputs
    - Color:
        - 1-4 channels
        - Integer (8-32), float
        - Special storage types: `GL_R3_G3_B2`, `GL_RGB10_A2`, …
    - Color updated on successful pass through all fragment tests

Depth buffer (Z-buffer)

- Contains depth information for each pixel
- Solves visibility problem
    - Geometry can be streamed
    - Works only for opaque objects
- Precision depends on:
    - z-buffer element type
    - projection - decreasing precision with increasing distance (choose proper near/far clipping planes)

Stencil buffer

- Additional buffer with integer elements
- Usually shares memory with z-buffer
- Limits area for rendering - stenciling
- Often used for shadow computation
- Can be updated by results of stencil and depth test
- Behavior setup:
    - `glStencilFunc`: what the test does
    - `glStencilOp`: what happens on test pass/fail

Operations and tests on fragments

- Scissor test
- Alpha test
- Depth test
- Stencil test
- Blending
- Dithering
- Logical operations (only integer based colors)

Depth test

- Different conditions for different objects (e.g. outline hidden objects)
- `glDepthFunc()`
    - `GL_NEVER`, `GL_ALWAYS`
    - `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, …
- Z-fighting - z-buffer precision
- `glPolygonOffset()`
    - Modulate z-value for specified primitives
- Early depth test optimization

Alpha test

- RGBA mode - fragment accepted/rejected by the alpha test
- void `glAlphaFunc(GLenum func, GLclampf ref);`
- Comparison function and reference value
- By default, ref is zero, func is `GL_ALWAYS`
- func: `GL_ALWAYS`, `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL GEQUAL`, `GL_GREATER` or `GL_NOTEQUAL`
- `glEnable(GL_ALPHA_TEST);`

Color blending

- How the color of the pixel is updated by fragment shader output
- Render transparent objects -
    - disable depth test, painters algorithm (order primitives)
    - order independent transparency - depth peeling
- `glBlendFunc()` mixing colors based on their respective alpha values
- The source color: the color of the fragment be drawn
- The destination color: the color already present in the color buffer

Antialiasing

- Supersampling (SSAA)
  - Render in higher resolution
  - Show downsampled image - smoothing
- Multisampling (MSAA)
  - Multiple depth/stencil tests per pixel
  - Estimates fragment coverage - smoothing on edges

Render buffer vs. texture

Best buffer for framebuffer attachments? - Render buffer object: - contains image, which will not be sampled (read) - optimized as render target - support MSAA - Textures: - optimized for read access - can be used later in the rendering pipeline

Triple buffering and V-Sync

- V-Sync: new frame is rendered in sync with monitor refresh frequency (60-100 Hz)
- Double buffering + V-Sync - small interval when none of the buffers can be touched - delay, idle GPU
- Second backbuffer - no delays, highest possible framerate
- Meaningful only when refresh rate lower than maximal possible rendering framerate

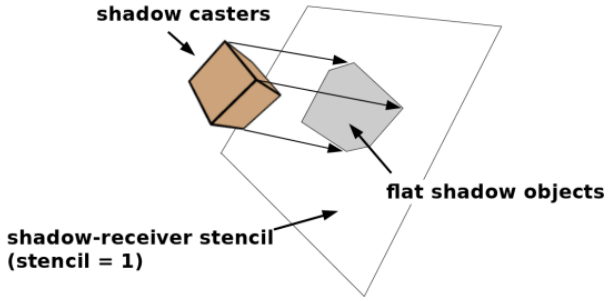
Shadows

Shadow casting

- Static Shadows: baked light/shadow map
- Dynamic shadows:
  - single shadow-receiving plane
    - simple approach, not generally usable
  - shadow mapping
    - shadow depth-buffer, supported in HW - shadowmap sampler
  - shadow volumes
    - precise but very computationally intensive
- sharp shadows (one pass)
- soft shadows (more passes, accumulation of results)

Shadow receiving plane

- sharp shadows - point light source
- use of stencil buffer and multiple scene passes
  - stencil prevents shadow duplication
- single shadow-receiving plane
- shadow could be opaque (destroying the original surface color) or transparent (only reducing the amount of light)



- **Procedure**

1. the whole scene rendered using ordinary projection
   - shadow-receiver sets stencil to 1
   - other objects zero this bit
2. potential shadow-casters rendered to the shadow-receiving plane
   - depth-test is off
   - special projection matrix
   - shadows are drawn only to the (stencil==1) pixels

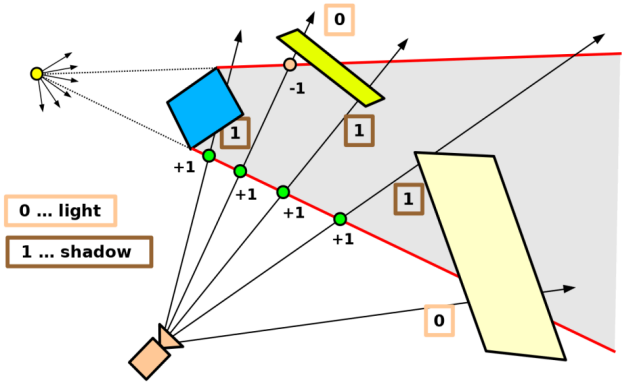
Face culling

- From the point of view of camera
- GPU can filter (face cull) according to vertex order:
  - glEnable(`GL_CULL_FACE`);

    - glFrontFace(`GL_CCW`);
    - glCullFace(`GL_BACK`); *// draw front faces only*
- Speed optimization

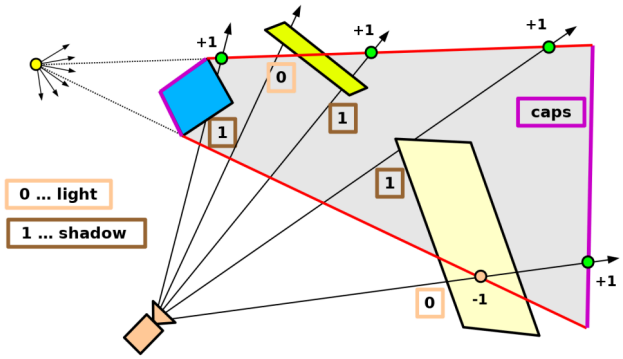
Shadow volume - depth pass

- shadow-caster - infinite shadow volume from countour (shadow solid)
- lateral faces of a shadow solid are considered, but invisible
- virtual ray from the camera is tested against these faces
- GPU can rasterize the virtual faces and “draw” them into the stencil buffer
    - Front faces increase stencil
    - Back faces decrease stencil
- stencil buffer values define shadows in the scene
- has to be done separately for each point light source



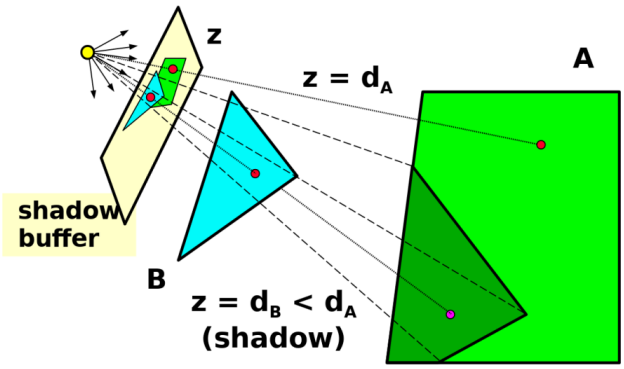
Shadow volume - depth fail

- Carmack’s reverse
- camera can be placed anywhere
- shadow solid sealed using “caps”: one is illuminated part of an object, the second one in infinity
- second phase: lateral shadow faces and both “caps”
    - Front faces - decrement on depth fail
    - Back faces - increment on depth fail
- third phase: stencil==0 means “light”



Shadow mapping

1. scene is rendered from the light-source viewpoint
    - no need to modify frame buffer, only depth-buffer has to be updated
2. depth-buffer is moved into a texture (“shadow map”)
    - regular projection according to the camera
    - use of projective texture coordinates
    - test actual distance of a fragment from the light source (in the world space) against shadow-map texture



- **Problems**
- Shadow acne
- Perspective aliasing
- Sharp shadows
- Hard to choose optimal size of shadow maps
    - Solution: cascaded shadow maps

Deffered Shading

- **Bottlenecks in rasterization pipeline**
    - Processing lots of lights
    - Complicated materials
    - Lots of fragments shaded and not used
- **Deffered shading**
    - Decouple geometry and light processing
    - Two stages:
        1. Render geometry to textures - multiple render targets (G-buffer)
        2. Posprocessing - apply light computations
- **Compositing step**
    - Compute shader or draw one fullscreen quad
    - Apply lighting for only visible fragments
    - All shading parameters come from uniforms and textures
    - Modern engines do postprocessing
        - Motion blur
        - Depth of field
        - Screen space ambient occlusion
        - Screen space decals
        - Bloom
        - HDR processing
- **Disadvantages**
    - Cannot handle transparency (depth peeling)
    - Complicated usage of multiple material types
    - Memory intensive
    - MSAA does not work:
        - Supersampling
        - Smoothing trick (small scale, rotate with linear interpolation,…)
        - Postprocessing - edge detection and masked smoothing, morphological AA (MLAA)

Effects

Surface details

Tangent space

- Local coordinate space
    - Z axis - normal N
    - X axis - tangent T (direction in which u coordinate changes)
    - Y axis - bitangent B (direction in which v coordinate changes)
- TBN matrix - transformation local tangent space to world space
- Orthonormal in texture space
- In general not othonormal in world space (only in special cases)
- **Mesh preprocessing**
    - Tangent space computed for each vertex

        - Triangle $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$, relative coordinates Q_2, Q_3, relative texture coordiantes $[s_2, t_2], [s_3, t_3]$:

$$\mathbf{Q}_i = \mathbf{P}_i - \mathbf{P}_1, \quad [s_i, t_i] = [\, u_i - u_1,\ v_i - v_1 \,], \quad i = 2, 3.$$

        - **Solve for tangent & bitangent:**

$$\mathbf{Q}_i = s_i\, \mathbf{T} \ + \ t_i\, \mathbf{B}, \quad i = 2, 3.$$

        - Then average $\mathbf{T}$ and $\mathbf{B}$ over all incident triangles (just like normals).
    - Approximation by orthonormal space:
        - Easy inverse matrix computation
    - Less data transferred to GPU - just $\mathbf{N}$ and a 4D tangent $(\mathbf{T}, w)$.
    - Passing normal and 4D tangent (w used for handedness determination)

$$\mathbf{B} = T_w(\mathbf{N} \times \mathbf{T}),$$

    > where T_w applies the chosen handedness sign $w \in \{\pm 1\}$.
- **Computation in fragment shader**
    - Current HW fast enough for on-the-fly computation
    - Fast enough to also compute inverse matrix (no need for orthogonalization)
    - How to compute differences from position and texture coordinates:

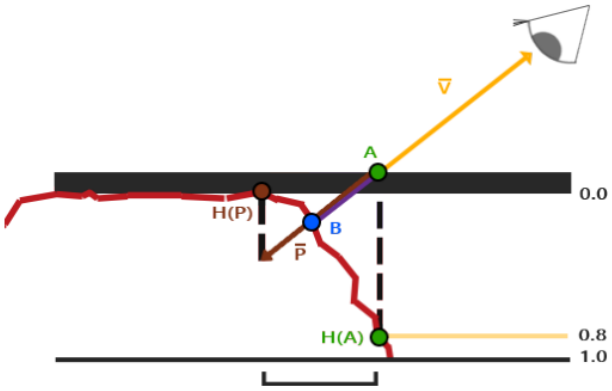
```
vec3 dp1 = dFdx(p);
vec3 dp2 = dFdy(p);
vec2 duv1 = dFdx(uv);
vec2 duv2 = dFdy(uv);
```

Bump mapping

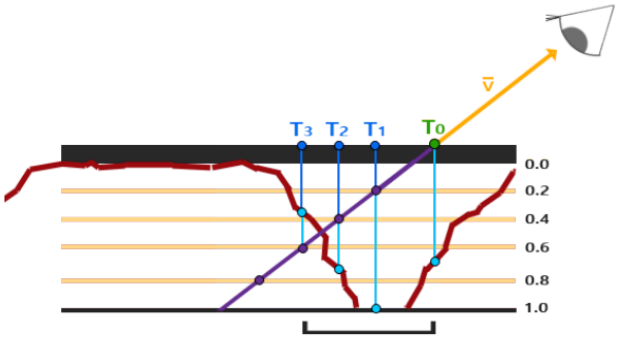
- Modulated normals in tangent space - normal map
    - normal $[0, 0, 1]$ mapped to RGB $[1/2, 1/2, 1]$
- Use TBN matrix to transform into world space for lighting computation

Parallax mapping

- Bump map - no parallax for surface displacement
- Effect can be simulated by modifying texture coordinates using displacement map
- **Basic**
    - Computation in local tangent space
    - Scale eye vector into $\mathbf{P}$ by $H(A)$
    - Crude estimation of texture offset $P_x y$
    - Problematic for steep displacements and low viewing angles

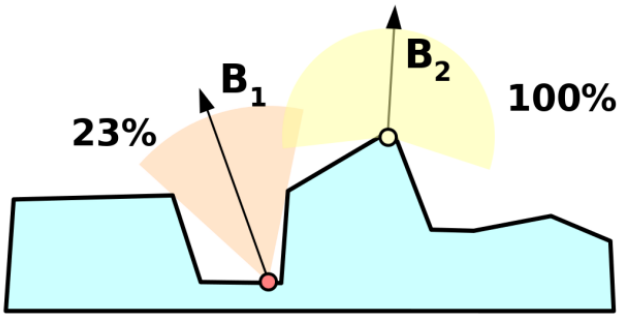


- **Steep parallax**
    - Better estimation of the texture offset
    - Check multiple layers to detect intersection more precisely



Ambient Occlusion

- constant *ambient term* not good enough
  - does not consider occlusion (even self-occlusion)
  - ridges are equally lighted as valleys
- pre-computed average (potential) contribution of surround light to the surface point
- for every surface point compute:
  - percentage of unoccluded rays from an environment (self-occlusion elimination) - *accessibility coefficient*
  - dominant light direction (best lit from) - B
  - technique: ray-casting from each point, counting rays without collision



- **Accessibility map utilization**
- accessibility coefficient
  - multiplication factor for ambient light approximation (instead of the k_A constant)
- dominant vector B
  - addressing for the environment light map
  - map should be blurred in advance
  - texture data are multiplied by the accessibility coefficient as well

Non-realistic rendering

X-Ray Vision

- Highlight invisible objects (occluded by different object)
- CAD system - invisible components
- VR, Games - highlight objects of interest
- Possible approaches:
  - Select occluded objects, render without depth test after everything else
    - Selection by different means
    - Problematic partial occlusion
  - Second render pass for highlighted objects, inverted depth test
    - Works with partial occlusion

Cartoon (Cel) Shading

- goal: results similar to human 2D graphics
  - contour emphasis
  - pen-and-ink drawing simulation (hatching)
  - imitation of painting techniques (oil, watercolor)
  - cartoon-style shading
- approaches (techniques)
  - special textures (coarse shading tones, ..)
  - procedural textures (fragment shader)
  - post-processing (for specific painting techniques)
  - $+$ combinations

Contour rendering

- No need for explicit definition of contours
- Solids have to be regular (closed)
- Two phases:
  1. front-facing faces only
     - no special rendering style
     - back-face culling
  2. edges of back-facing faces only
     - more thick line (`glLineWidth()`) - contour lines will stick out
     - alternative - render backfaces of blown-up mesh (no scaling)

Cartoon light model

- light model similar to “Blinn-Phong”
  - diffuse term $cos\alpha$
  - optional specular term $cos^h\beta$
- diffuse term indexes simple ramp texture, or quantize the intensity
  - only small number of color tones
  - no texture filtering for sharp outlines
  - CAD applications - determination of plane orientation
- optional specular term with priority
  - thresholding for white-color highlight

Postprocessing

Basic postproccessing operators

- Process outputs from deffered shading stage
- Texture coordinate transformation
- Spatial filtering - operations on pixel (texel) neighborhood
  - Linear filtering - convolution
    - Edge detection
    - Smoothing
    - Bluring
    - Bloom
  - Non-linear:
    - Morphological operations
    - Median filtering

Coordinate transform

- Transform input u, v coordinates ($f : [0, 1]^2 \rightarrow [0, 1]^2$)
- Warping
- Optical Effects
  - Fish eye lens
  - Barrel distortions
- Extreme stretching limited by number of texels
- Higher order interpolation - bicubic,…

Spatial filtering

- Value of the pixel is updated by some function over the neighboring pixels
- Linear combination - convolution
  - Mask containing weights (kernel)
- Nonlinear operations - min, max, median, …
- **Implementation**
  - Fragment shader:
    - u_{step}, v_{step} - single texel offset in normalized texture space
    - `texelFetch()` - access via non-normalized coordinates
  - Compute shader:
    - Better optimization options

Gaussian smoothing

- Gaussian distribution (normal) - result of combined random processes
- Used for smoothing (bluring), noise reduction
- σ determines kernel radius - $68 - 95 - 99.7$ rule

- Separable filter:
  - Equivalent to two pass filtering with horizontal and vertical 1D kernel
- $2n$ instead of n^2 texture reads

Contour (edge) detection

- Edges in image - sharp changes in value
- Places with high gradient
- Alternative for cartoon shading
- **Numerical differentiation**
  - Finite difference:

$$\frac{f(x+h) - f(x)}{h}$$

  - Symmetric (central) difference:

$$\frac{f(x+h) - f(x-h)}{2\,h}$$

  - Higher-order schemes improve numerical stability but at the cost of wider stencils > Note: Differentiation amplifies high-frequency noise
  - Discrete gradient kernels (3-point stencil):

$$D_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, \qquad D_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

- **Sobel filter**
  - Numerical partial derivations with small smoothing
  - Gradien magnitude - edge strength
  - Threshold small values - filter out small fluctuations

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- **Discontinuities in G-Buffers**
  - Z-buffer
    - Boundary between objects
    - Different parts of objects
  - Normals
    - Strong edge without normal interpolation
    - Boundary between objects
  - ID-buffer (stencil)
    - Boundaries only between different objects
- **Combined contours**
  - Detected discontinuities in normals and depth
  - Summ together - all important contours together

Advanced texturing

Bottlenecks of modern renderers

- Memory transfers between CPU (RAM) and GPU
- Communication with driver:
  - Fixed pipeline:
    - Lots of API calls to manage state
  - OpenGL 3.0+:
    - Bind operations
    - Setting shader uniforms
    - Draw calls
- Multi-material scene/object
  - Changing shader programs + repeated uniform setup
  - Bind new textures on material switch
  - Multiple draw calls

Uniform buffer objects

- Advantages:
  - Same uniforms in multiple shader programs:

```
uniform vec4 camera_position;
uniform vec4 light_position;
uniform vec4 light_diffuse;
```

  - Single buffer cointaining the data
  - Larger uniform storage
  - Faster switching for uniform blocks
  - Switch to uniform block in GLSL

```
uniform shader_data
{
vec4 camera_position;
vec4 light_position;
vec4 light_diffuse;
};
```

  - C++ counterpart:

```
struct shader_data_t
{
    float camera_position [4];
    float light_position [4];
    float light_diffuse [4];
} shader_data ;
```

  - Create uniform buffer:

```
GLuint ubo = 0;
glGenBuffers(1, &ubo);
glBindBuffer(GL_UNIFORM_BUFFER, ubo);
glBufferData(GL_UNIFORM_BUFFER, sizeof(shader_data), &shader_data, GL_DYNAMIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

  - Update data:

```
glBindBuffer(GL_UNIFORM_BUFFER, gbo);
GLvoid*
p = glMapBuffer(GL_UNIFORM_BUFFER, GL_WRITE_ONLY);
memcpy(p, &shader_data, sizeof(shader_data))
glUnmapBuffer(GL_UNIFORM_BUFFER);
```

  - Connect UBO and GLSL program:

```
block_index = glGetUniformBlockIndex(program , "shader_data");
GLuint binding_point_index = 2;
glUniformBlockBinding(program, block_index, binding_point_index);
...
glBindBufferRange(GL_UNIFORM_BUFFER, binding_point_index, gbo, 0, sizeof(shader_data_t));
```

Bindless textures

- How to prevent texture binding?
  - Generate integer *handle* for each texture:
    - from texture object alone
    - from texture object and sampler
    - from specific image within texture
  - Texture state becomes immutable (can update contents)
  - Access texture by handle from shaders
    - cannot be used until made *resident*
  - Safety: errors may crash the GPU, program, OS
  - Extensions: `ARB_bindless_texture`, `NV_bindless_texture`

Sparse virtual texture

- Also known as *megatextures* (Idsoft - Rage)
- Different approach to binding prevention - one large texture for whole scene
- Texture may be larger than GPU memory (over-subscription)
  - Similar to virtual address space and physical memory
  - Pages are texture tiles
  - Page table for translation of texture coordinates
- Each object in scene uniquely textured
  - Artist less limited by technical aspects
- **Page mapping**

    - Access the page table with original texture coordinates (nearest neighbor)
    - No special coordinate mapping
    - Within-page offset:
        - Depends on mip-map level
    - Feedback analysis
        - Separate pass - render page IDs (low resolution)
        - Determine pages + mip-map levels
        - Loading missing pages - delay before used (mip-map fallback)

Decals, Billboards

- Runtime interaction with the scene
- Additional details:
    - Bullet holes
    - Graffiti
    - Local material weathering
    - Footsteps

Approaches

- Megatextures:
    - Draw decals directly in the scene texture
    - Maybe permanent without increased overhead
- Special geometry rendered in front of the object
    - Z-fighting, depth offset
    - Simple scene - textured quad
    - Geometry projection in general case
    - Adding decals increases scene complexity - only few latest/important kept
- Screen space decals - deferred shading

Projecting geomtry

- Oriented bounding box:
    - projector along z-axis
    - x,y are mapped to u,v coordinates
- Intersection with scene geometry
    - select intersecting triangles
    - cut triangles - project to projector space, uv-mapping

Screen space

- Deffered shading
- Render projector box
    - Reject fragments which project outside the box (use z-buffer + view direction)
    - Flattened box - projected on the geometry
- Normal mapping:
    - Normal buffer may contain modulated normals
    - Underlying geometry normal - partial derivatives in the z-buffer
- Problems:
    - Clipping the projector box
    - Projection on 90 degree corners

Billboards

- Billboard - semitransparent texture showing more complicated object/scenery
    - texture is usually mapped on a rectangle
    - often perpendicular to view direction
    - … following the viewer - special transform matrix
    - rotation around vertical axis only (unsightly from above)
- usage
    - trees and bushes (even unoriented billboards, multi-billboards)
    - complex inscriptions, 2D graphics, HUD, lens flare..

Impostors

- Impostor - billboard created dynamically (as necessary) in a rendering engine
    - cache of complex scenery (not very dynamic)
    - complex object/scenery (geometric or color complexity)
    - for distant objects mostly
    - hierarchy, LoD, multiple instances of the (almost) same object…

- trees, bushes
  - impostors might be oriented along main branches..
- technique: HW render-target textures

Noise functions

- Critical for realistic textures and models
- Simplifies creation of natural variations
- Applications: terrain, procedural texturing, simulations
- Key for realism in visual effects and games

Functions

- Generate pseudo-random
- Smooth gradients - frequency limited
- Controlled randomness mimics natural forms
- Types:
  - Value
  - Gradient (Perlin, Simplex)
  - Cellular (Worley)
  - Fractal Noise

Perlin noise

- Developed by Ken Perlin, 1983
- Algorithm:
  - Gradient vectors computed at grid points
  - Interpolated across grid to produce smooth transitions
- Properties:
  - Visually isotropic in 2D and 3D
  - Repeats over large scales, which can be controlled
- Applications: Terrain, clouds, fire textures

Simplex noise

- Ken Perlin, 2001
- Algorithm:
  - Similar to Perlin but with simplex grid (triangular/hexagonal)
  - Reduces computational complexity, especially in higher dimensions
- Properties:
  - Faster computation and lower complexity than Perlin
  - Scales more efficiently to higher dimensions (4D and beyond)
- Avoids square-grid artifacts of Perlin noise

Worley noise

- Steven Worley, 1996
- Algorithm:
  - Points randomly distributed, partitioned into cells
  - Noise generated based on proximity to nearest points
- Properties:
  - Produces a voronoi diagram-like appearance
  - Can simulate phenomena like cracked surfaces, sponge textures
- Applications: Stone, water effects, organic textures

Compositing noise functions

- Combines multiple noise types to increase texture complexity
- Techniques:
  - Layering different scales and amplitudes
  - Masking layers to control influence areas
- Example: Mix Perlin (base texture) + Worley (detail enhancement)
- Enhances detail and realism in procedural content

Volumetric effects

- Light usually passes through some medium (air, water, . . .)
- Intensity, color (polarization) may be modulated:
  - Attenuation (fog)
  - Scattering (sunbeams, blue sky)

- Simulated by:
  - Ray traversal
  - Blending billboard slice planes

Ray casting

- Space traversal along light ray
- Integrating properties along the ray:

$$\nu = \int_{\text{ray}_{\text{start}}}^{\text{ray}_{\text{end}}} f(s)\,\mathrm{d}s$$

- Discrete samples:
  - Regular voxel grid
  - Procedural description
- Numerical integration:
  - Piece-wise constant
  - Interpolation (linear, polynomial)
  - …

Sunbeams

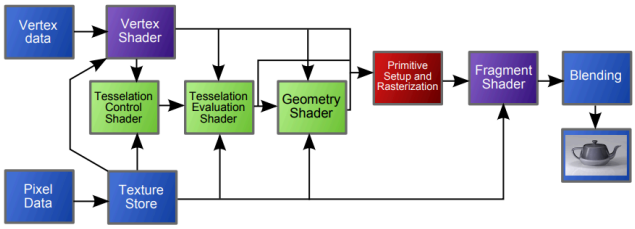
- Also known as crepuscular rays, god rays, …
- Scattering on particles under direct light:
  - Sun + clouds
  - Point light source + dusty room
- **Implementation**
  - Deffered shading
  - Ray casting from viewer to each pixel
    - Ray sampling
    - Check if sample illuminated - shadow map test
    - Apply light scattering (physical model) to illuminated points
    - Aggregate the effect and apply to color buffer
  - Heavy computation
    - Downsampled g-buffer
    - Bluring result to prevent aliasing
- **Other approaches**
- Create light volume geometry from shadow map and light source
  - Solve the rendering integral in intervals defined by light mesh
- Screen space approach:
  - Directional light source bluring (decreasing alpha)
  - Ligth source must be in the image

Generate Geometry on GPU

Geometry instancing

- Lots of models, same vertex data
  - Particle systems
  - Forrests
  - Armies
  - . . .
- Different transformations, texture data, . . .
- Many draw calls - bottleneck
  - Rendering - fast
  - Issuing draw commands - slow
- Instancing - draw multiple objects by single call
- Replace standard draw calls by instanced versions (extra count parameter)
  - `glDrawArraysInstanced()`
  - `glDrawElementsInstanced()`
- In shaders build-in variable `gl_InstanceID`
  - From interval $[0, count)$
  - Used for indexing arrays of offsets, shifting texture coords, . . .

Tessellation shaders



Why

- Adaptive subdivision based on a variety of criteria (size, curvature, etc.)
- Coarser models, but finer ones displayed (geometric compression)
- Detailed displacement maps without supplying equally detailed geometry
- Adapt visual quality to the required level of detail

Patch primitives

- New PATCH primitive
- Fixed number of vertices:
  - `glPatchParameteri( GL\_PATCH\_VERTICES, num );`
  - Structure defined by shader implementor

Shader organization

- Tessellation Control Shader (TCS)
  - Computes tessellation levels (fixed, distance to eye, screen space, hull curvature, . . .)
  - One invocation per output vertex
- Tessellation Primitive Generator (TPG)
  - Fixed-function
  - Generates predefined patterns in u-v-w barycentric coordinates
- Tessellation Evaluation Shader (TES)
  - Evaluates the surface in uvw coordinates
  - Interpolates attributes
  - Applies displacements

Lots of technical details, it just makes new LODs…

Geometry shaders

- last *optional* shader stage before rasterizer
- similar to tesselation
  - generating new geometry
  - similar results by different principles
  - new primitives are generated directly in shader (no fixed primitive generator)

Input

- assembled primitives (no *strips*, *loops*, or *fans*)
  - points . . . 1
  - lines . . . 2
  - lines_adjacency . . . 4
  - triangles . . . 3
  - triangles_adjacency . . . 6
- Access to information about the whole primitive
- New primitives with adjacency information

Output

- possible output primitives:
  - points
  - line_strip
  - triangle_strip
- types of input and output primitives are **independent**
  - input forgotten after shader execution
- output **0** or **more** primitives (up to implementation defined limit)

Application

- Shadow volume

- Use triangles with adjacency:
  1. Render front cap
     - Pass through illuminated faces
  2. Render back cap
     - Same polygons projected to infinity (depth clamping)
  3. Render extruded silhouette
     - extrude edges separating illuminated and shadowed faces (compare $dot(N, L)$)

Mesh shaders

- Flexible and scalable geometry processing
- Replaces Vertex + Geometry Shaders
- Works with small **meshlets** instead of individual triangles
- Improves **GPU parallelism** and reduces draw calls
- Converges towards **compute shader-like workflow**
- Supported by **NVIDIA Turing/RTX** and **AMD RDNA2 GPUs**
- Available in **DirectX 12 Ultimate** and **Vulkan 1.2+**

Why

- Tessellation and geometry shaders are limited:
  - Tessellation in function
  - Geometry in speed
- Mesh shaders - convergence to compute shaders
  - Better hardware saturation
  - More flexible

Scientific Visualization

Volumetric data

Data representation

- Regular, 3-dimensional grid of samples (voxels)
  - Scalar values - density, absorption coefficients, event counting
  - Vectors
  - Color
- 3D texture:
  - Trilinear filtering
  - Easy slicing in general direction
- 2D texture:
  - Set of textures
  - Texture atlas
  - Manual filtering in Z-direction

Volume Rendering Integral

$$I(D) = I_0 \, \exp\Big(-\int_{s_0}^{D} \kappa(t) \, \mathrm{d}t\Big) \; + \; \int_{s_0}^{D} q(s) \, \exp\Big(-\int_{s}^{D} \kappa(t) \, \mathrm{d}t\Big) \, \mathrm{d}s$$

- s_0: entry point
- D: exit point (camera position)
- $q(s)$: emission at point s
- I_0: initial intensity at s_0 (background emittance)
- $\kappa(t)$: absorption coefficient

Viewport aligned slices

- Generate proxy geometry
  - Viewport aligned slices (billboards)
  - Limited by volume bounding box - limit fragment count
  - Convex - easy to triangulate
- Enable framebuffer blending
  - Color attachment with float precision

Ray-casting

- Generate rays from camera through each pixel
  - Fragments generated by rendering bounding volume
- Discrete samples along the ray
- Numerical computation of the rendering integral

Volume compositing schemes

- In Direct Volume Rendering, compositing accumulates color and opacity along the viewing ray.
- Two main compositing orders:
  - **Front-to-back**: processes samples from the eye toward the volume.
  - **Back-to-front**: processes samples from deep in the volume toward the eye.

Compositing equations

Color bleeding

Maximum intensity projection

Direct volume rendering

Combined geometry rendering

- Opaque geometry
  - Rendered before volume
  - Rays terminated by value in z-buffer
- Transparent geometry
  - Checking for geometry/ray intersections during ray traversal
  - Color computed together with volume sampling

Transfer functions

1D Transfer functions

- Runtime fuzzy classification
- Transfer function $g(v) : R \rightarrow R^4$
- Ray sample: $g(f(x)$
- Maps scalar value to RGBA color.
- Implementation:
  - 1D RGBA texture with interpolation
  - Final sample color - access TF texture

Gradient

1D Transfer functions + light

- Better surface shape perception.
- Compute shading for opaque regions (α channel over some threshold)
- Normalized gradient as surface normal.

2D Transfer functions

Gradient magnitude

- Can be computed on the fly.
- Ability to separate borders from homogeneous regions.

Improvements

- Lighting
- Jittering
- Speedup techniques

Isosurfaces

- Polygonal mesh representing level set
- Volume preprocessing:
  - Cuberille (+filtering)

    - Marching cubes, tetrahedra, . . .
- Use normal rasterization pipeline for rendering
- Ray-casting
    - Search for isovalue crossings
    - Fine search in subintervals for intersection point
    - Gradient for surface normal

Vector fields

Data sources

- Physical simulations:
- Fluid dynamics
- Particle simulations
- Electromagnetic fields (Maxwell)

Numerical integration

- Simulate motion under vector field influence
- Numerical integration
    - Euler method - low numerical stability, fast
    - Higher order Runge-Kutta methods

Glyphs, icons, probes

- rendering glyphs
    - Large number of similar geometries
    - Instanced rendering
        - Impostors for complicated geometries
    - Geometry shader:
        - From point samples generate glyph geometry

Line integral convolution

- Underlying texture blurred along vector directions
    - Multiple texture accesses in fragment shader - integration
- on surface (Compute in object fragment shader)

Points clouds

Data sources

- Surface points:
    - 3D scanner output
    - Scene reconstruction:
        - Stereo cameras
        - Camera + depth sensor (Kinect)
        - Single moving camera
- Random spatial samples:
- Unstructured vector field
- Unstructured volume

Rendering

- Glyph for each point
    - Colored/textured facets
- Glyph for group of points
    - Size, shape - properties of point group
- Unstructured volume samples:
    - Datastructure for fast queries (octree, . . .)
    - Ray sample - weighted average of points in certain radius
- Surface reconstruction:
    - Distance field
    - Isosurface rendering

Speedup Techniques, Other APIs

Lots of stuff, read the slides

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 ## LODs

GPGPU

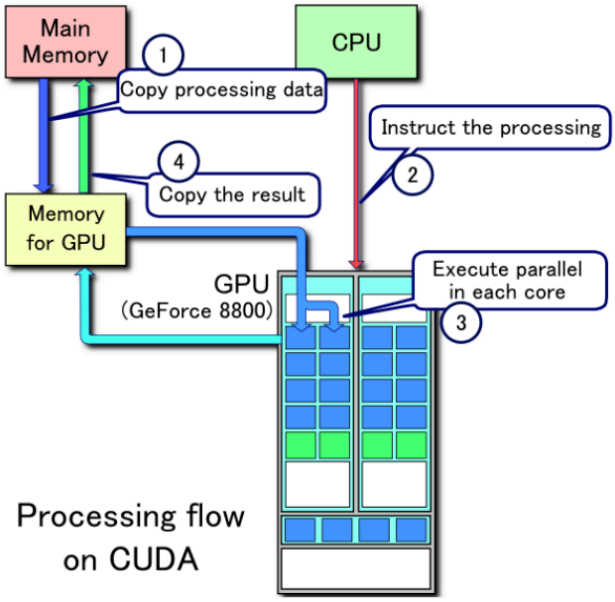
- parallelizable tasks on GPU
  - code size and memory are the only limits
- many core computing

CUDA (Compute Unified Device Architecture)

- CUDA GPU: group of highly threaded **streaming multiprocessors** (SM)
- each SM has a set of **streaming processors** (SP) - CUDA cores
- SP within one SM share control circuits, instruction decoder and instruction cache
- SIMT (Single Instruction Multiple Threads) - warp

Program structure

- GPU code = kernel
  - deployed on thousands of threads
  - GPU threads are much more light-weighted (thread creation, ctxsw a couple of machine cycles)



- Keywords `__global__`, `__host__`, `__device__`
  - where the code can run
  - from where it can be called

Kernel execution

```
MyKernel<<<gridSize , blockSize , dynamicSharedMemorySize , streamID>>>(arg1 , arg2 , . . . ) ;
```

- gridSize
  - int or dim3
  - Specifies 3D structure of thread blocks
- blockSize
  - int or dim3
  - Specifies 3D structure of threads in block
- dynamicSharedMemorySize
  - amount of dynamically allocated shared memory
  - 0 is default
- streamID

    - Which stream is used for execution
    - 0 is default
- Usual approach - map grid and blocks on input data
- Block:
    - Executed on single SM
    - Cannot be removed until finished
- Threads in **warp**:
    - Set of threads that all share the same code
    - Follow the same execution path (masking execute on branching)

Streams

- CUDA Stream - queue of commands (kernel execution, memory transfers, event)
- Commands in stream serialized
- Different streams - possible concurrency
- Default stream 0 always exists (can be per thread)
- `cudaStreamCreate()`
- Synchronization:
    - `cudaStreamSynchronize(stream)`
    - Event system

Memory types

Host (RAM)

- Normal memory RAM
- By default cannot be accessed from device
    - Must be copied to device memory

Global memory

- Actual GPU memory
- Used as normal linear memory - pointer arithmetics
- Management:
    - `cudaMalloc()`, `cudaMallocPitch()`, `cudaMalloc3D()`
    - `cudaMemcpy()`, `cudaMemcpyToSymbol()`
- Read from kernel can take hundreds of cycles

Texture memory

- Allow usage of texturing HW:
    - Spatial caching
    - Filtering
- Limited by predefined element types (colors)
    - No custom structures

Shared memory

- Same space as L1 cache
    - Division is customizable
- `__shared__` keyword
- Shared by threads in block
- Use when same value access is multiple times in block execution (not necessarily by same thread)

Registers

- 32-bit registers
- Divided between active warps
- Shared memory + registers limit occupancy:
    - Number of active warps vs. max possible warps on SM
- Limiting number of registers may lower performance
    - May be necessary to run at least 1 block on SM

Advanced features

Unified memory

Dynamic parallelism

- Kernels can be executed from kernels on device
- Parent kernel waits until children finishes
- Allows adaptive thread execution

Parallel algorithms

- Algorithms for massively parallel architectures:
  - Often bottom up design
  - Shallow datastructures
  - Memory access patterns considered first
- Try to make all operations local only
- Problem reformulation:
  - Search for possible constrains
  - Solve dual problem
  - Cellular automata
  - . . .

Compute shaders

Usage

- Write compute shader in GLSL
  - Define memory resources
  - Write `main()` function
- Initialization
  - Allocate GPU memory (buffers, textures)
  - Compile shader, link program
- Run it
  - Bind buffers, textures, images, uniforms
  - Call `glDispatchCompute(...)`

Deep neural networks

- Another neural networks renaissance
- Large neural networks with lots of layers
  - Convolutional networks
- Large numbers of identical neurons - highly parallel by nature
- Backpropagation
  - Millions of parameters
  - Large training set
- Training vs. inference

OpenCL

- Code structure similar to shader programming

Realtime raytracing

Terminology

Whitted's raytracing

- Extending raycasting by using recursion
- New reflected/refracted rays are generated and color is recursively propagated back
- Perfect reflections/refractions
- Simple shadow computation - ray to light source

Distributed raytracing

- Modeling soft effects:
  - Soft shadows
  - Depth of field
  - Soft reflections
- Shooting multiple rays
  - Sampling the domain (angle, lens)
  - Weighted average of the payloads
  - Suppression of alias

Path raytracing

- Systematic approach to solving global illumination problem
- Rendering equation:
  - Integral equation in which the radiance leaving a point is given as the sum of emitted plus reflected radiance under a geometric optics approximation
- Raytracing using Monte-Carlo method for the rendering equation solving
- Bidirectional path tracing - faster convergence

Shaders

Ray tracing engines utilize various specialized shaders to achieve realistic rendering. The primary shaders used are: - Ray Generation Shader - Intersection Shader - Any-Hit Shader - Closest-Hit Shader - Miss Shader - Callable Shader

Nvidia RTX

- Bounding Volume Hierarchy (BVH) Traversal
- Ray/Triangle Intersection Testing
- Parallel Processing

Shaders

- New GLSL shader types:
  - Any hit
  - Intersection
  - Miss
  - Closest hit
- All shaders must be available - ray may intersect any object

BLAS (Bottom-Level Acceleration Structures)

- Purpose:
  - Represent individual objects or meshes in the scene.
- Structure:
  - Consist of geometric primitives like triangles.
  - Optimized using a Bounding Volume Hierarchy (BVH).
- Usage:
  - Built once and reused for multiple frames.
  - Can be updated if the geometry changes.

TLAS (Top-Level Acceleration Structures)

- Purpose:
  - Represent the entire scene, including instances of **BLAS**.
- Structure:
  - Contains instances of BLAS with their transformations.
  - Organized in a BVH for efficient traversal.
- Usage:
  - Built from instances of BLAS.
  - Updated when the scene layout changes.

Optix

- OptiX is not just a raytracer
- OptiX is framework for creating application using raytracing, independent from any specific method
- Build on CUDA architecture
- Most of the components programmable
- Usable not only for CG, but also for:
  - collision detection
  - visibility determination
  - sound propagation simulation
  - volume estimation of complicated objects
  - . . .
- Abstract model of generic raytracer
- Future-proofed - build to scale with future development of powerful GPUs
- Similar abstraction to OpenGL/DirectX/Vulcan
- Mechanism for execution of custom CUDA C code
  - Shading + recursive rays
  - Camera model, ray generator
  - Ray payload

    - Intersection with arbitrary geometry (e.g. exact sphere without tesselation)
    - . . .

Programs

- 8 types of GPU programs supported by OptiX
- **Ray Generation** - Launch entry point, invoked per-pixel/sample
- **Exception** - Called on invalid state (e.g., stack overflow)
- **Closest Hit** - Executed on closest geometry hit (used for shading)
- **Any Hit** - Invoked on all intersections (used for early-out, transparency, shadows)
- **Intersection** - User-defined geometry intersection logic (procedural primitives)
- **Miss** - Executed when ray misses the scene (e.g., skybox sampling)
- **Direct Callable** - Fast inlined function calls via SBT (e.g., shading models)
- **Continuation Callable** - May trigger new ray traces or shading logic

Execution

- All informations and data passed to context instance
- Specify dimensions and execution parameters
- Ray generator is executed, once for each element(pixel)
- Results stored into output buffer

Denoising

CSG, Depth peeling, Transform feedback

Transform feedback

- Captures output from Vertex/Tessellation/Geometry Shaders
- Bypasses the rasterizer
- Writes data directly into GPU buffer objects

When to use

- Particle systems
- GPU-based simulations
- Geometry manipulation and reuse
- General-purpose GPU computation

Depth peeling

- An Order-Independent Transparency (OIT) technique
- Renders transparent objects correctly without sorting
- Captures multiple depth layers via multiple rendering passes

Why

- Traditional alpha blending is order-dependent
- Geometry sorting fails for intersecting or dynamic shapes
- Depth peeling ensures visual correctness in transparency

Workings

- Pass 1: Capture nearest depth layer
- Pass 2+: Use previous depth to discard closer fragments
- Final: Blend all layers for final image

Requirements

- Multiple framebuffers (for color and depth)
- Shaders to compare and discard previous depth
- Blending enabled (typically premultiplied alpha)

CSG (Constructive solid geometry)

- Used by CAD applications
- Set operations on geometry primitives

    - Union
    - Intersection
    - Subtraction

Animation

Vertex Animation

- Vertex animation involves the manipulation of individual vertices to create movement and deformation of 3D models.
- Typically used for animating complex deformations and morphing effects.
- Unlike skeletal animation, vertex animation directly modifies the positions of vertices.

Keyframe interpolation

Morph targets (Blend shapes)

Skinning

- Skinning is a method used for character animation where a mesh (skin) is deformed based on the movement of an underlying skeleton (bones).
- Essential for creating realistic character movements.
- Allows for complex deformations driven by skeletal structures.

Linear blend skinning (LBS)

Dual quaternion skinning (DQS)

Rigid skinning

Physics based animation

- Physics-based animation uses physical laws to simulate realistic movements and interactions in real-time.
- Adds realism to animations by mimicking real-world physics.
- Commonly used for particles, rigid bodies, fluids, cloth, and hair.

Particle systems

- Simulate phenomena like fire, smoke, and explosions.
- Each particle represents a small part of the effect.
- Behavior governed by forces such as gravity, wind, and collision.
- Efficiently handled on the GPU for real-time performance.

Rigid body dynamics

- Simulate the motion of solid objects.
- Objects can move, rotate, and collide with each other.
- Governed by Newton’s laws of motion.
- Used for simulating objects like bouncing balls, falling debris, etc.

Fluid simulations

- Create realistic water, liquid, and other fluid animations.
- Techniques include SPH (Smoothed Particle Hydrodynamics) and grid-based methods.
- Computationally intensive but can be optimized for real-time using the GPU.

Cloth simulation

- Simulate the behavior of fabric as it moves and interacts with objects.
- Techniques include mass-spring systems and finite element methods (FEM).
- Used for realistic clothing, curtains, and other fabric materials.

Hair simulation

- Simulate individual strands or clumps of hair.
- Techniques include particle-based methods and volumetric approaches.
- Ensures realistic movement and interactions with wind, gravity, and collisions.

Animation blending

- Ensures smooth transitions between animations, enhancing realism.
- Prevents abrupt changes in movement that can break immersion.
- Allows for dynamic and responsive character behaviors.

Linear blending

Non-Linear blending

Additive blending

Inverse kinematics

- Inverse Kinematics (IK) is a technique used to calculate the necessary joint angles to achieve a desired position for a part of a character, such as a hand or foot.

pandoc zkouska.md -s -o zkouska.html –katex –css=style.css