Graphics 2014

Advanced Rasterization
Announcements

Extra tutorials next week
- The usual times: Thu 15:15h – 17:00h
- Rooms: BBL 023, BBL 079, BBL 083 (not BBL 165)

Questions + Answers
- Please mail me your questions
  - Will be passed on to tutors
  - Best: mail before end of this week
- Preparation for final exam
3D Rendering Steps

Geometric Model

Perspective

Visibility

Local Illumination

Smooth Shading

Advanced Rasterization

Global Illumination
Topics

Supplementary Details

- Projective geometry
- Rasterization and clipping
- Transformations & Normals

Texture Mapping

- Basic idea
- Perspective correction
Topics

Advanced Texture Mapping

- Aliasing, Filtering & Mipmapping (short)
- 2D and 3D Textures
- Shadow maps
- Displacement maps
- Bump mapping / normal maps
- Environment Maps
- Image-based Lighting
Topics

Modern Rasterization Pipeline

- Vertex and Pixel Shaders
  - Extensions
- Render targets
  - Color buffers
  - Float buffers
  - Stencil buffer
- Textures
  - 2D & 3D textures
  - Cube maps
Addendum
Projective Geometry

advanced topics
main ideas
Constructing Projective Spaces

Projective Space $\mathbb{P}^d$:
- Euclidian ("affine") space $\mathbb{R}^d$ embedded in $\mathbb{R}^{d+1}$
- At $w = 1$
- Identify all points on lines through the origin
  - *Representing* the same Euclidian point

$p' = \{(wp) \mid w \in \mathbb{R}^\neq 0\}$

$p' \in \mathbb{P}^1$

$p' \in \mathbb{P}^2$
Constructing Projective Spaces

Translators:
- Sheering of the projective space
  \[
  \begin{pmatrix}
  1 & 0 & t_x \\
  0 & 1 & t_y \\
  0 & 0 & 1
  \end{pmatrix}
  \]
- Translation of the embedded affine space
Normalization

Conversion between

- Cartesian coordinates (Euclidian space)
- Homogeneous coordinates (projective space)

\[ \mathbf{x} \to \begin{pmatrix} w \mathbf{x} \\ w \end{pmatrix} \]

Normalization\(^*\)

\[ \frac{1}{w} \mathbf{x} \leftarrow \begin{pmatrix} \mathbf{x} \\ w \end{pmatrix} \]

\(^*\) overloaded name
do not confuse with \( \mathbf{x}/\|\mathbf{x}\| \)
Mathematical Language

Form equivalence classes

- \((x \equiv y) \iff \exists \lambda \in \mathbb{R}: (x = \lambda y)\)
- Think of overloading `operator=()`

Even more formally (math students)

- Consider group of uniform scalings
  
  \[
  G = \left\{ \begin{pmatrix} \lambda & \vdots & 0 \\ 0 & \ddots & \vdots \\ 0 & \cdots & \lambda \end{pmatrix} \middle| \lambda \neq 0 \right\}
  \]
- Symmetry group of the representation:
  
  \[
  P^d = \mathbb{R}^d \mod G
  \]
  - Ignore “irrelevant information”
Properties

Projective Maps

- Linear maps in the higher dimensional space
- Scale at any time:
  \[ y = M \cdot x \equiv \frac{M \cdot x}{x \cdot w} \equiv \frac{M \cdot x}{y \cdot w} \quad \text{(for } w \neq 0) \]

- Why? Scaling yields the same point!
Important:

- We have: \( x \equiv \lambda x \) (for \( \lambda \neq 0 \))
- But in general: \( x + y \neq x + \lambda y \)
  - For correct result: Normalize first (same \( w \))
Vectors & Points

Interpretation

- Points: \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}, w \neq 0
- Vectors: \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} – “pure directions”
Vectors & Points

Rules

- Subtracting points yields vectors
  - Normalize first!
- Vectors can be added to
  - Other vectors
  - Points (normalize first!)
Rasterization and Clipping
How to rasterize Primitives?

Two problems
- Rasterization
- Clipping
Rasterization

Assumption

- Triangles only
- Triangle not outside screen
- No clipping required
Triangle Rasterization

Several Algorithms...
Triangle Rasterization

Example: two slabs (tutorials)
Triangle Rasterization

\[ \Delta x \text{ constant} \]

precompute and
add in each step

Incremental rasterization
Incremental Rasterization

Precompute steps in x, y-direction

- For boundary lines
- For linear interpolation within triangle
  - Colors
  - Texture coordinates (more later)

- Inner loop
  - Only one addition ("DDA" algorithm)
  - Floating point value
  - Strategies
    - Fixed-point arithmetics
    - Bresenham / midpoint algorithm (requires if; problematic on modern CPUs)
Rasterization

How to rasterize Primitives?

Two problems

- Rasterization
- Clipping
Why Clipping?

Crashes – write to off-screen memory!
Clipping Strategies

Pixel Rejection

- “if (x,y $\notin$ screen) continue;”
  - Can be arbitrarily slow (large triangles)
  - Nope. Not a good idea.

Screen space clipping

- Modify rasterizer to jump to visible pixels
  - See tutorial 5
- Efficient
- Still problems with when crossing camera plane ($w = 0$) $\Rightarrow$ a semi-good idea
Smart Slab Renderer

Does not crash, optimal complexity

- $O(k)$ for $k$ output fragments
Problem:

- Triangles crossing camera plane!
  - Wrong results
- Need object space clipping

\[ y' = f \frac{y}{z} \]
View Frustum Clipping

- Near clipping plane
- Far clipping plane
- Four side planes
- Six planes clip triangles against all six planes
Incremental Algorithm
Incremental Algorithm
Incremental Algorithm

Output: Multiple Triangles
Further Optimization

View Frustum Culling

- Complex shapes (whole bunnies)
- Coarse bounding volume (superset)
  - Cube, Sphere
  - Often: Axis-aligned bounding box
- Reject all triangles inside if bounding volume outside view frustum
Transformations & Normals
Remark 1: Scene Graphs

(a) two objects

(b) instantiation

(c) hierarchical instantiation
Animation

Hierarchical Animation

- Rotate wheels
- Move car with rotating wheels

“Kinematic Chains”

- Body, upper arm, lower arm, hand, fingers,...
- Relative transformations handled correctly automatically

(c) hierarchical instantiation
Implementation

Data Structure
- Simplest version: Tree
- Instancing: Directed Acyclic Graph (DAG)

Algorithm
- Depth-first-traversal
- Multiply transformation nodes
- Use associativity to order
  - Matrix stack to store intermediate results
- \( \left( (M_1 \cdot M_2) \ldots M_{n-1} \right) \cdot M_n \)

\[\text{deepest (applied first)}\]
\[\text{topmost (visited first)}\]
Remark 2: Transforming Normals

How to transform normals of a surface?

Three cases

- Translations
  - Do not apply to normals!

- Orthogonal transformations
  - Rotations, reflections
  - Transform normals and points the same way

- General linear transformations
  - Points: $\mathbf{p'} = \mathbf{Mp}$
  - Normals: $\mathbf{n'} = (\mathbf{M}^T)^{-1}\mathbf{n}$
Explanation

Implicit plane equation
\[ \langle \mathbf{n}, \mathbf{p} \rangle = \mathbf{n}^T \cdot \mathbf{p} = d \]
- \( \mathbf{p} \) is a vector
- \( \mathbf{n} \) is a co-vector

Change of coordinates:
- \( \mathbf{p} \rightarrow \mathbf{M} \cdot \mathbf{p} \)
- \( \mathbf{n}^T \rightarrow (\mathbf{M}^{-1} \cdot \mathbf{n})^T \)

Result: same plane
\[ (\mathbf{M}^{-1} \cdot \mathbf{n})^T \cdot \mathbf{M} \cdot \mathbf{p} = \mathbf{n}^T(\mathbf{M}^{-1} \cdot \mathbf{M}) \cdot \mathbf{p} = \mathbf{n}^T \cdot \mathbf{p} = d \]
Texture Mapping
Texture Mapping

Idea:

- Map image to triangle
- Additional details
- Hard to model with geometry
- Much cheaper than fine geometric tessellation
Texture Coordinates

Define Mapping to Image

- Texture coordinates at vertices
- In between: linear interpolation
- Defines an affine map
2D Texture Mapping
Define Mapping to Image

- Texture coordinates at vertices
- In between: affine ("linear") interpolation
- Defines an affine map

Technically, this is an affine map, but people often call it "linear interpolation"
Affine Map

- Map coordinate system \( \{ \mathbf{p}_1, (\mathbf{t}_1, \mathbf{t}_2) \} \) to \( \{ \mathbf{p}_1, (\mathbf{t}_1', \mathbf{t}_2') \} \)

\[
\mathbf{x} \rightarrow \mathbf{p}_1 + \begin{pmatrix} \mathbf{t}_1' & \mathbf{t}_2' \\ \mathbf{t}_1 & \mathbf{t}_2 \end{pmatrix} \begin{pmatrix} \mathbf{t}_1' & \mathbf{t}_2' \\ \mathbf{t}_1 & \mathbf{t}_2 \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{x} \mathbf{p}_1' \end{pmatrix}
\]
Rasterization

- Project vertices
  - Keep texture coordinates as specified
- Create fragments
  - Lookup texture color

Texture Lookup
lookup color for each fragment
**Rasterization: Inverse Map**

- $t_2 = p_3 - p_1$
- $t_1 = p_2 - p_1$

**Affine Map**

- Map coordinate system $\{p_1, (t'_1, t'_2)\}$ to $\{p_2, (t_1, t_2)\}$

$$x \rightarrow p'_1 + \left( \begin{array}{c} t'_1 \\ t'_2 \end{array} \right) \cdot \left( \begin{array}{c} t_1 \\ t_2 \end{array} \right)^{-1} (x - p_1)$$
Texture Mapping

Texture space to screen space:

\[ \textbf{x} \rightarrow \textbf{p}_1 + \left( \begin{array}{cc} \textbf{t}_1 & \textbf{t}_2 \end{array} \right) \cdot \left( \begin{array}{cc} \textbf{t}'_1 & \textbf{t}'_2 \end{array} \right)^{-1} (\textbf{x} - \textbf{p}'_1) \]

Screen space to world space:

\[ \textbf{x} \rightarrow \textbf{p}'_1 + \left( \begin{array}{cc} \textbf{t}'_1 & \textbf{t}'_2 \end{array} \right) \cdot \left( \begin{array}{cc} \textbf{t}_1 & \textbf{t}_2 \end{array} \right)^{-1} (\textbf{x} - \textbf{p}_1) \]

*) Formally: this is a change of coordinate system
Barycentric Coordinates

- 2D coordinate system (in plane)
- Triangle edge coordinates
- Same as ratio of area of opposing triangle to overall triangle area
Barycentric Coordinates

Interpretation:

\[ \mathbf{x} \rightarrow \mathbf{p}_1' + \begin{pmatrix} \mathbf{t}_1' & \mathbf{t}_2' \end{pmatrix} \cdot \begin{pmatrix} \mathbf{t}_1 & \mathbf{t}_2 \end{pmatrix}^{-1} (\mathbf{x} - \mathbf{p}_1) \]

- Transform to barycentric coordinates, then to texture coordinates
Non-uniform spacing!

- 2D texture mapping will create artifacts!
Example
Example

2D Texture Mapping

3D Texture Mapping

Obviously, we want this!
A Related Problem

2D Texture Mapping
triangulation dependent

two linear maps glued together

3D Texture Mapping
triangulation independent

nonlinear map ("homography")
Perspective Correction

Incorrect results:
- Linear*) interpolation in screen space

Correct results
- Linear*) interpolation in object space
- Uneven steps in texture coordinates between pixels

How to compute?

*) again, this is actually an affine map, but the term “linear interpolation” is almost exclusively used here
Correct Perspective Texturing

Two solutions

- **Absolute computation**
  - Setup ray equation for each pixel
  - Compute ray-triangle intersection
  - Possible and correct, but slow

- **Incremental interpolation**
  - Do not interpolate $u, v$ but $\frac{u}{z}, \frac{v}{z}$
    - Gives correct results in screen space!
  - Multiply by $z$ in the end
    - Interpolate $\frac{1}{z}$ in screen space ($z$ also non-linear!)
    - Divide by $\frac{1}{z}$ (approximation strategies for speed)

(Lazy “option” #3: use GPU/GfxLib and don’t worry :) )
3D Triangles

Non-uniform spacing!

depth values (z)
3D Triangles

proportional:

\[ z(y) \sim v(y) \]

\[ \frac{1}{z} \sim y + c \]
3D Triangles

\[ \mathbf{v}(\mathbf{y}) = \mathbf{m}_v \cdot \mathbf{y} + \mathbf{v}_0 \]

\[ \mathbf{r}(\mathbf{z}) = \mathbf{m}_r \cdot \mathbf{y} \cdot \mathbf{z} \]

\[ \mathbf{z} = \frac{\mathbf{v}_0}{(\mathbf{m}_r \cdot \mathbf{y} - \mathbf{m}_v)} \]

\[ \mathbf{v}(\mathbf{y}) = \frac{\mathbf{m}_v \cdot \mathbf{v}_0}{(\mathbf{m}_r \cdot \mathbf{y} - \mathbf{m}_v)} + \mathbf{v}_0 \]

\[ \left[ \frac{1}{\mathbf{z}} \right](\mathbf{y}) = \frac{1}{\mathbf{v}_0}(\mathbf{m}_r \cdot \mathbf{y} - \mathbf{m}_v) \]

\[ \left[ \frac{\mathbf{v}}{\mathbf{z}} \right](\mathbf{y}) = \mathbf{m}_r \cdot \mathbf{y} \]

affine

linear
Aliasing and Anti-Aliasing
Aliasing

simple sampling

anti-aliasing (Gaussian)
Aliasing

**Minification:** Moiré

*Sampling aliasing*

**Magnification:** “Staircasing”

*Reconstruction aliasing*
Magnification

In hardware:

- Bi-linear*) interpolation
- Linear blend in $u$- and $v$-direction

*) same here, it is a (bi-) affine map, but called “(bi-) linear” in literature / APIs
Magnification

In hardware:
- Bi-linear* interpolation
- Linear blend in $u$- and $v$-direction

*) you know, linear ~ affine...
Sampling Aliasing

- Sampling grid misses information, spacing too big
- Creates Moiré (or noise, if unstructured)
Solution

- Average over neighborhood
  - Heuristic: “leave no free space”
- Best: weighted filters with overlap (e.g. Gaussians)
Graphics Hardware

Two Steps

Moderate Minification

- Bilinear*) interpolation

Strong Minification

- Mip-mapping
  - Average 2 × 2 pixels
  - Store reduce size by 1/2
  - Iterate

- Trilinear interpolation

*) same here
Summary

**Minification**
- Average multiple texels

**Magnification**
- Average over multiple pixels

**Swap screen & texture**
The Full Story: Fourier Transforms

(a) a continuous function and its frequency spectrum
(b) a regular sampling pattern (impulse train) and its frequency spectrum
(c) sampling: frequencies beyond the Nyquest limit $\nu_s/2$ appear as aliasing
(d) reconstruction: filtering with a low-pass filter $R$ to remove replicated spectra

- more in “advanced graphics” -
Topics

Texture mapping variants

- 3D Textures
- Shadow maps
  - Ambient Occlusion
- Environment Maps
  - Image-based Lighting
- Bump mapping / normal maps
- Displacement maps
3D Textures

- Use 3D array of “voxels”
- u,v,w-coordinates
- Texture space itself
Shadow Maps

Create shadow map
- Render scene from light source
- Store depth buffer

Render scene from camera
- Project fragment to depth buffer/light source
  - If occluder in front → dark
  - Otherwise → bright
Shadow Maps Pitfalls

Offset problem

- Camera pixels (slightly) different from light pixels
- Need small offset for depth comparison

Aliasing

- Visible staircasing
- Light projection ≠ screen pixels

Spot-lights only

- → Cubemaps (later)
Resolution

low resolution

medium resolution
Resolution

high resolution

very high resolution
Offset Problem

good offset

bad offset
Ambient Occlusion

Average of 256 Images

light sources randomly sampled on enclosing sphere
Ambient Occlusion

Average of 2560 Images
light sources randomly sampled on enclosing sphere
Environment Maps

Approximate Reflections
- Store panoramic image (“360°”) of environment
- Use for reflection

Approximation
- Far away environment
- Single bounce
- No occlusion in path
- Refraction less accurate (single bounce?)
Implementation: Cube-Maps

Cube maps

- Cube with 6 textured faces
- “Infinite size”

Cube-map texture lookup

- Very easy!
- Arbitrary vector
- \( \text{texcube} \left( \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right) \)
  1. \( \text{arg max} \{ |x|, |y|, |y| \} \)
  2. sign
  3. divide by \(| \cdot |\)

- Select face: largest entry, sign
- 2D coordinates: divide by maximum entry
Cube Map Lookups

$|y|$ largest $\rightarrow$ horizontal

$y > 0$ $\rightarrow$ right face

$x$ $\div$ $y$ $\rightarrow$ texture coordinate
Rendering with Natural Light

Traditional Cube Maps
- Created using rendering
- 6 passes with different camera settings
- Latest hardware: “render to cubemap”

Rendering with Natural Light
- Use HDR photographs for cube maps
  - High-dynamic range is important
- Diffuse, specular, glossy \(\Rightarrow\) filter (blur) cube map
Title-Bunny

Ambient Occlusion with Natural Light

adaptive sampling density ~ intensity

texture (c) Paul Debevec, USC
Bump-Maps / Normal Maps

Normal Maps

- Store normal in texture
- Map to tangent coordinate frame
- Need normals $\mathbf{n}$ and tangent field $\mathbf{t}$
- Then: coordinate transform
Bump Maps

- Same as normal maps
- But only height field given
- Need to precompute normals

Precompute Normals

- Discrete partial derivatives
  \[ \Delta x := \frac{f(x + 1, y) - f(x)}{h} \]
  \[ \Delta y := \frac{f(x, y + 1) - f(x)}{h} \]
  \[ n \approx \begin{pmatrix} -\Delta x \\ -\Delta y \\ 1 \end{pmatrix} \]
Displacement Maps

Simplest Method
- Start with Bump Map
- Create actual mesh by displacement in normal direction

Needs powerful hardware
- Fancy in the past few years for real-time / games
- Offline rendering used this ever since
Hardware Architecture
void shade() {
    doThis();
    doThat();
}

program

Execution Unit

ALU / FPU

ALU / FPU

ALU / FPU

ALU / FPU

... frequent read/writes ...

Stream buffer
(small, fast)

little bandwidth

few read/writes,
large chunks

Cache (small)

Main Memory
(big, high latency, limited bandwidth)
Simplified GPU Model

A GPU is a vector processor

- SIMD – single instruction, multiple data
- “Branching” possible at costs
  - Divergent instructions executed serially (write masks)

Specifically: Stream processor

- Memory interface main problem
- Idea: big on-chip buffer, work within buffer
- Write to/from memory more rarely
  - Direct access still possible (“texture fetches”)
  - Caching, hide latency using multi-threading
Advantages

High throughput

- GeForce GTX Titan Z: 8 TFlops (32-bit SP - 2 chips)
- Radeon R9 295x2: 11 TFlops (32-bit SP - 2 chips)
- Xeon Phi 7120: 2.5 TFlops (32-bit SP)
- Dual Xeon E5-2697: 500 GFlops (32-bit SP)

Not all workloads

- Parallel problems
  - Not all problems are parallelizable!
- Instruction stream coherence
  - Additional constraints over MIMD computers!

*) example numbers! architectures have different advantages
Hardware Architecture

Programmable Vertex Shaders

- **Input: Vertex Buffer**
  - Multiple attributes
  - Position, color, normals, texture coordinates, etc...

- **Execute program**
  - Texture reads possible

- **Output: one vertex per vertex**
  - One-in-one-out
  - Internal queue (efficiency)
Rasterizer

- Clipping of triangles
- Creation of fragments
- Interpolate attributes
  - With perspective correction!
- Hard-wired for efficiency
- Output: Long sequence of fragments
Pixel Shader

Pixel shader

- Input: Fragment with interpolated attributes
- Perform computation
  - Arithmetics
  - Texture reads: tex2D, tex3D, texCube, ...
  - Includes filtering (anti-aliasing)
- Output:
  - Color (always)
  - Alpha-value (optional)
  - Depth (optional, reduces efficiency; no early fragment rejection)
Final Combiner

Write to frame-buffer

- Depth test and update
- Color update
  - Overwrite, additive, subtractive, alpha-blending (and a few more)
- (Usually) hardwired
Recent Extensions

Geometry shader

- Between vertex shader and rasterizer
  - Convert single primitive into a small number of additional ones
  - Amount limited (e.g., 32 vertices output)

Hull shader / tessellation shader

- Better adaptive subdivision
  - Spline surfaces
  - Displacement mapping