CSE 167: Introduction to Computer Graphics Lecture #17: Deferred Rendering

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2018

Announcements

- TA evaluations
- CAPE evaluation
- Final project blog entries due:
  - Tonight , Dec 4th at 11:59pm
  - Next Tuesday, Dec 11th at 11:59pm
- Video due:
  - Wednesday, Dec 13th at 12 noon

Lecture Overview

- Deferred Rendering
  - Deferred Shading
  - Bloom and Glow
  - Screen Space Ambient Occlusion

Deferred Rendering

- Opposite to Forward Rendering, which is the way we have rendered with OpenGL so far
- Deferred rendering describes post-processing algorithms
  - Requires two-pass rendering
  - First pass:
    - Scene is rendered as usual by projecting 3D primitives to 2D screen space.
    - Additionally, an off-screen buffer (G-buffer) is populated with additional information about the geometry elements at every pixel
      - Examples: normals, diffuse shading color, position, texture coordinates
  - Second pass:
    - An algorithm, typically implemented as a shader, processes the G-buffer to generate the final image in the back buffer

Deferred Shading

- Postpones shading calculations for a fragment until its visibility is completely determined
  - Only visible fragments are shaded
- Algorithm:
  - Fill a set of buffers with common data, such as diffuse texture, normals, material properties
  - Render lights with limited extent and use data from the buffers for the lighting computation
- Advantages:
  - Decouples lighting from geometry rendering
  - Several lights can be applied with a single draw call. E.g., >1000 lights can be rendered at 60 fps
- Disadvantages:
  - More expensive (memory, bandwidth, shader instructions)
- Tutorial:
  - http://gamedevs.org/uploads/deferred-shading-tutorial.pdf

Particle system with glowing particles. Source: Humus 3D

UCSD

Lecture Overview

- Deferred Rendering Techniques
  - Deferred Shading
  - Bloom and Glow
  - Screen Space Ambient Occlusion

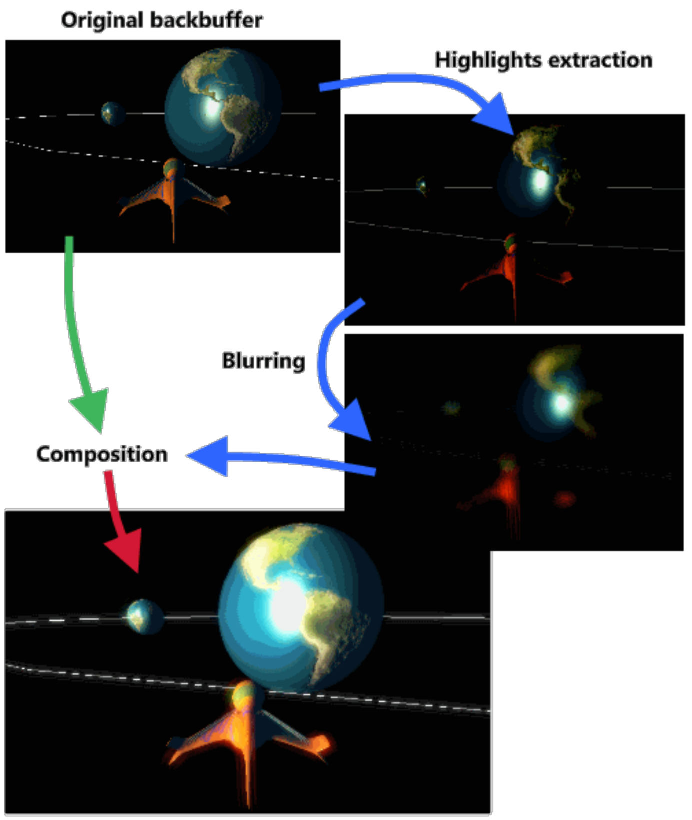
Bloom Effect

Left: no bloom, right: bloom. Source: http://jmonkeyengine.org

- Computer displays have limited dynamic range
- Bloom gives a scene a look of bright lighting and overexposure
- Provides visual cues about brightness and atmosphere
  - Caused by light scattering in atmosphere, or within our eyes

Bloom Shader

- Step 1: Extract all highlights of the rendered scene, superimpose them and make them more intense
  - Operates on G-buffer
  - Often done with G-buffer smaller (lower resolution) than frame buffer
  - Highlights found by thresholding luminance
- Step 2: Blur off-screen buffer, e.g., using Gaussian blur
- Step 3: Composite off-screen buffer with back buffer



Bloom shader render steps.
Source: http://www.klopfenstein.net

Glow vs. Bloom

- Bloom filter looks for highlights automatically, based on a threshold value
- If you want to have more control over what glows and does not glow, a glow filter is needed
- Glow filter adds an additional step to Bloom filter: instead of thresholding, only the glowing objects are rendered
- Render passes:
  - Render entire scene back buffer
  - Render only glowing objects to a smaller off-screen glow buffer
  - Apply a bloom pixel shader to glow buffer
  - Compose back buffer and glow buffer together

Video: Glowing Lava

- https://www.youtube.com/watch?v=hmsMk-skquI

References

- Bloom Tutorial
  - http://prideout.net/archive/bloom/
- GPU Gems Chapter on Glow
  - http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch21.html
- GLSL Shader for Gaussian Blur
  - http://www.ozone3d.net/tutorials/image_filtering_p2.php

Lecture Overview

- Deferred Rendering Techniques
  - Deferred Shading
  - Glow
  - Screen Space Ambient Occlusion

Screen Space Ambient Occlusion (SSAO)

- "Screen Space" → deferred rendering approach
- Approximates ambient occlusion in real time
- Developed by Vladimir Kajalin (Crytek)
- First use in PC game Crysis (2007)

SSAO component

UCSD

Ambient Occlusion

- Crude approximation of global illumination
- Often referred to as "sky light"
- Global method (not local like Phong shading)
  - Illumination at each point is a function of other geometry in the scene
- Appearance is similar to what objects appear as on an overcast day
  - Assumption: concave objects are hit by less light than convex ones

Basic SSAO Algorithm

- First pass:
  - Render scene normally and write z values to G-buffer's alpha channel
- Second pass:
  - Pixel shader samples depth values around the processed fragment and computes amount of occlusion, stores result in red channel
  - Occlusion depends on depth difference between sampled fragment and currently processed fragment

Ambient occlusion values in red color channel
Source: www.gamerendering.com

UCSD

SSAO With Normals

- First pass:
  - Render scene normally and copy z values to G-buffer's alpha channel and scene normals to RGB channels
- Second pass:
  - Use normals and z-values to compute occlusion between current pixel and several samples around that pixel

No SSAO

With SSAO

SSAO Discussion

- Advantages:
  - Deferred rendering algorithm: independent of scene complexity
  - No pre-processing, no memory allocation in RAM
  - Works with dynamic scenes
  - Works in the same way for every pixel
  - No CPU usage: executed completely on GPU
- Disadvantages:
  - Local and view-dependent (dependent on adjacent texel depths)
  - Hard to correctly smooth/blur out noise without interfering with depth discontinuities, such as object edges, which should not be smoothed out

SSAO References

- Nvidia's documentation
  - http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf

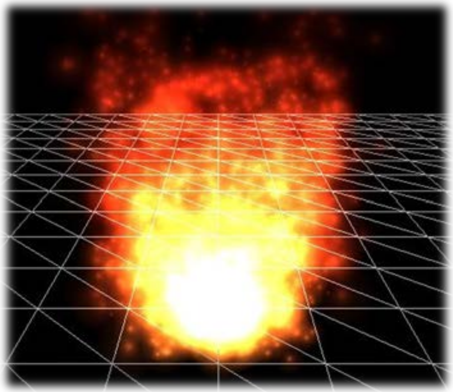
Lecture Overview

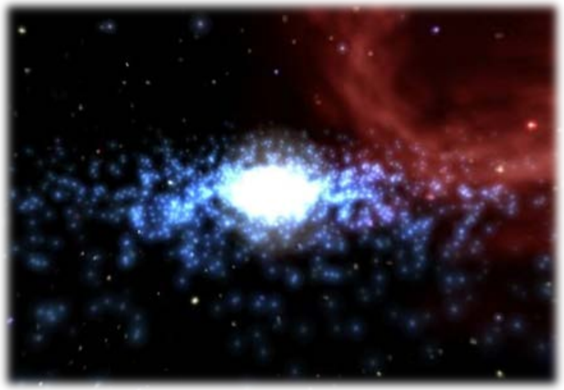
- Particle Systems
- Collision Detection
- Bump Mapping

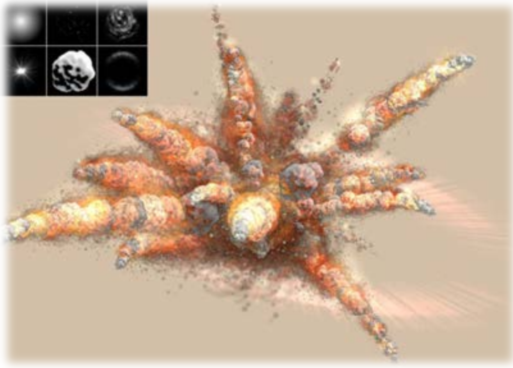
Particle Systems

Particle Systems

- Used for:
  - Fire/sparks
  - Rain/snow
  - Water spray
  - Explosions
  - Galaxies





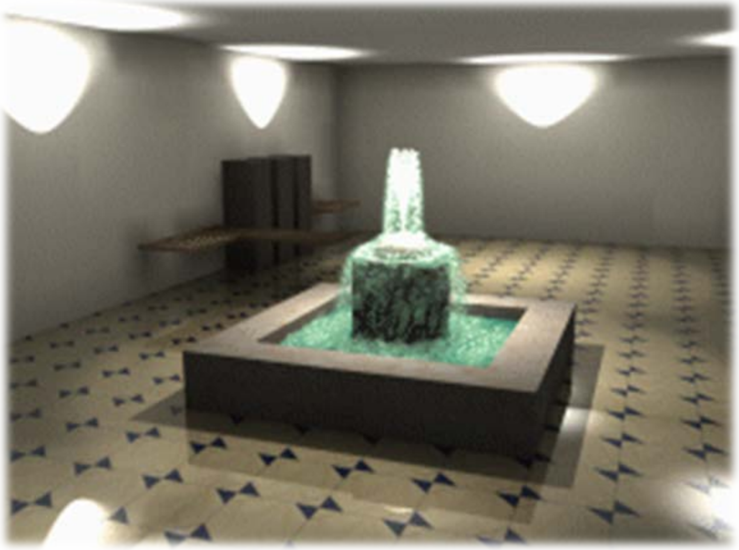


Internal Representation

- Particle system is collection of a number of individual elements (particles)
  - Controls a set of particles which act autonomously but share some common attributes
- Particle Emitter: Source of all new particles
  - 3D point
  - Polygon mesh: particles' initial velocity vector is normal to surface
- Particle attributes:
  - position (3D)
  - velocity (vector: speed and direction)
  - color + opacity
  - lifetime
  - size
  - shape
  - weight

Dynamic Updates

- Particles change position and/or attributes with time
- Initial particle attributes often created with random numbers
- Frame update:
  - Parameters: simulation of particles, can include collisions with geometry
    - Forces (gravity, wind, etc) accelerate a particle
    - Acceleration changes velocity
    - Velocity changes position
  - Rendering:
    - GL_POINTS
    - GL_POINT_SPRITE
    - Point shader



Source: http://www.particlesystems.org/

Point Rendering – Vertex Shader

```
uniform mat4 u_MVPMatrix;
uniform vec3 u_cameraPos;

// Constants (tweakable):
const float minPointScale = 0.1;
const float maxPointScale = 0.7;
const float maxDistance   = 100.0;

void main()
{
    // Calculate point scale based on distance from the viewer
    // to compensate for the fact that gl_PointSize is the point
    // size in rasterized points / pixels.
    float cameraDist = distance(a_position_size.xyz, u_cameraPos);
    float pointScale = 1.0 - (cameraDist / maxDistance);
    pointScale = max(pointScale, minPointScale);
    pointScale = min(pointScale, maxPointScale);

    // Set GL globals and forward the color:
    gl_Position  = u_MVPMatrix * vec4(a_position_size.xyz, 1.0);
    gl_PointSize = a_position_size.w * pointScale;
    v_color      = a_color;
}
```

Demo

- Particle system in WebGL:
  - http://nullprogram.com/webgl-particles/

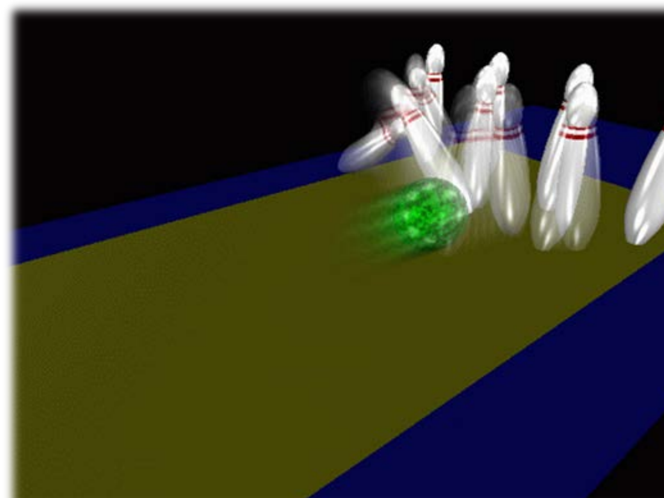
References

- Tutorial with source code by Bartlomiej Filipek, 2014:
  - http://www.codeproject.com/Articles/795065/Flexible-particle-system-OpenGL-Renderer
- Articles with source code:
  - Jeff Lander: "The Ocean Spray in Your Face", Game Developer, July 1998
    - http://www.darwin3d.com/gamedev/articles/col0798.pdf
  - John Van Der Burg: "Building an Advanced Particle System", Gamasutra, June 2000
    - http://www.gamasutra.com/view/feature/3157/building_an_advanced_particle_.php
- Founding scientific paper:
  - Reeves: "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects", ACM Transactions on Graphics (TOG) Volume 2 Issue 2, April 1983
    - https://www.evl.uic.edu/aej/527/papers/Reeves1983.pdf

Collison Detection

Collision Detection

- Goals:
  - Physically correct simulation of collision of objects
    - Not covered here
  - Determine if two objects intersect
- Slow calculation because of exponential growth $O(n^2)$:
  - # collision tests = n*(n-1)/2



Intersection Testing

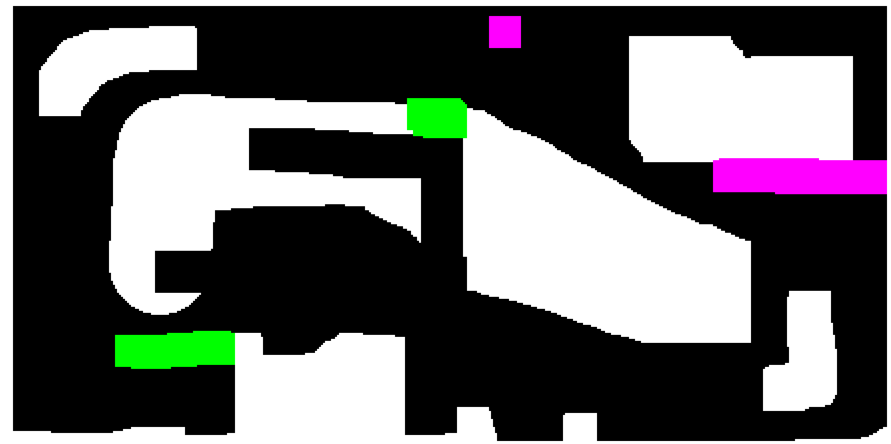
- Purpose:
  - Keep moving objects on the ground
  - Keep moving objects from going through walls, each other, etc.
- Goal:
  - Believable system, does not have to be physically correct
- Priority:
  - Computationally inexpensive
- Typical approach:
  - Spatial partitioning
  - Object simplified for collision detection by one or a few
    - Points
    - Spheres
    - Axis aligned bounding box (AABB)
  - Pairwise checks between points/spheres/AABBs and static geometry

Sweep and Prune Algorithm

- Sorts bounding boxes
- Not intuitively obvious how to sort bounding boxes in 3-space
- Dimension reduction approach:
  - Project each 3-dimensional bounding box onto the x,y and z axes
  - Find overlaps in 1D: a pair of bounding boxes can overlap if and only if their intervals overlap in all three dimensions
    - Construct 3 lists, one for each dimension
    - Each list contains start/end point of intervals corresponding to that dimension
    - By sorting these lists, we can determine which intervals overlap
    - Reduce sorting time by keeping sorted lists from previous frame, changing only the interval endpoints

Collision Map (CM)

- 2D map with information about where objects can go and what happens when they go there
- Colors indicate different types of locations
- Map can be computed from 3D model, or hand drawn with paint program
- Granularity: defines how much area (in object space) one CM pixel represents



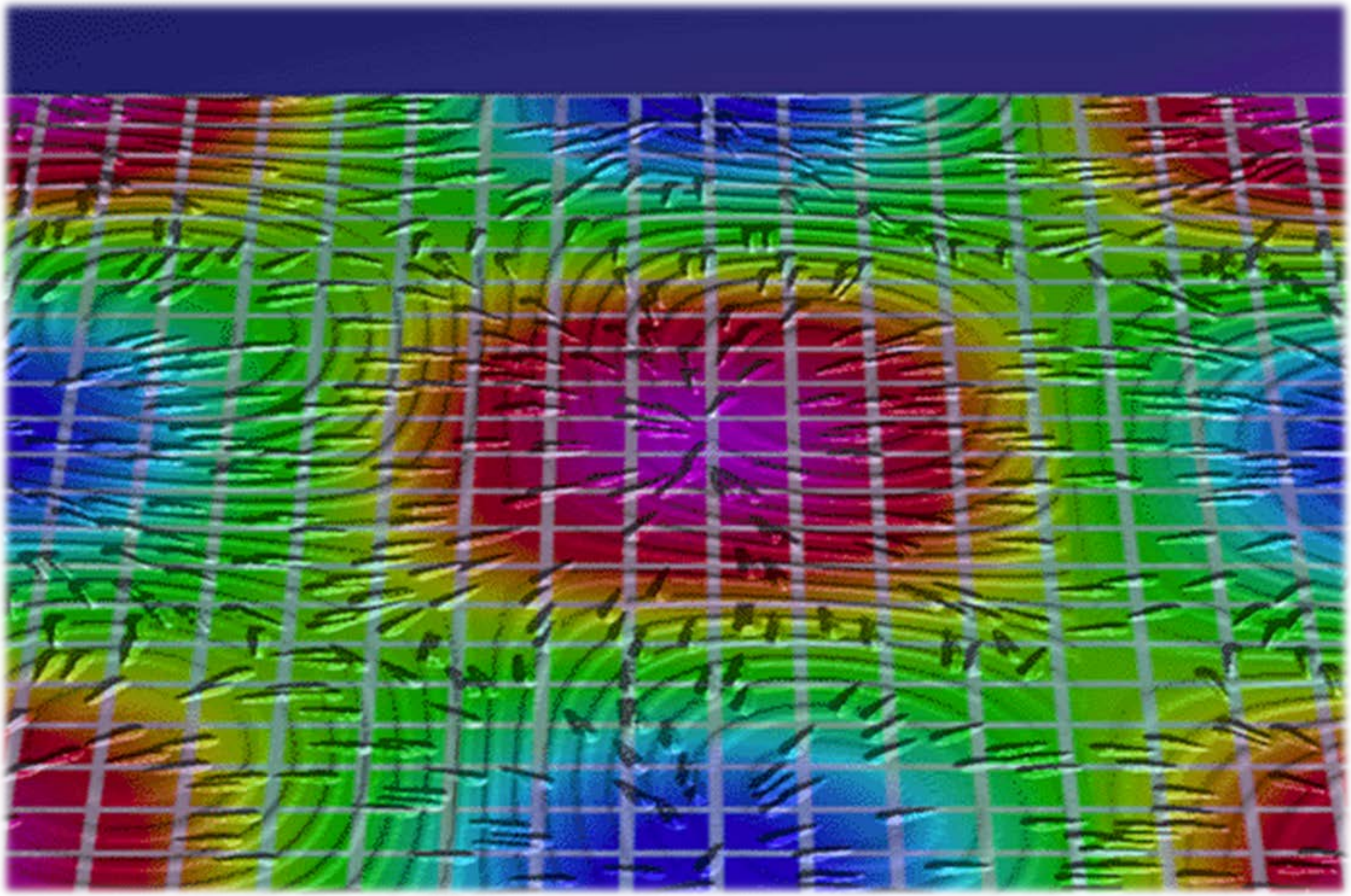
Bump Mapping

Bump Mapping

- Many textures are the result of small perturbations in the surface geometry
- Modeling these changes would result in an explosion in the number of geometric primitives.
- Bump mapping attempts to alter the lighting across a polygon to provide the illusion of texture.

[This chapter includes slides by Roger Crawfis]

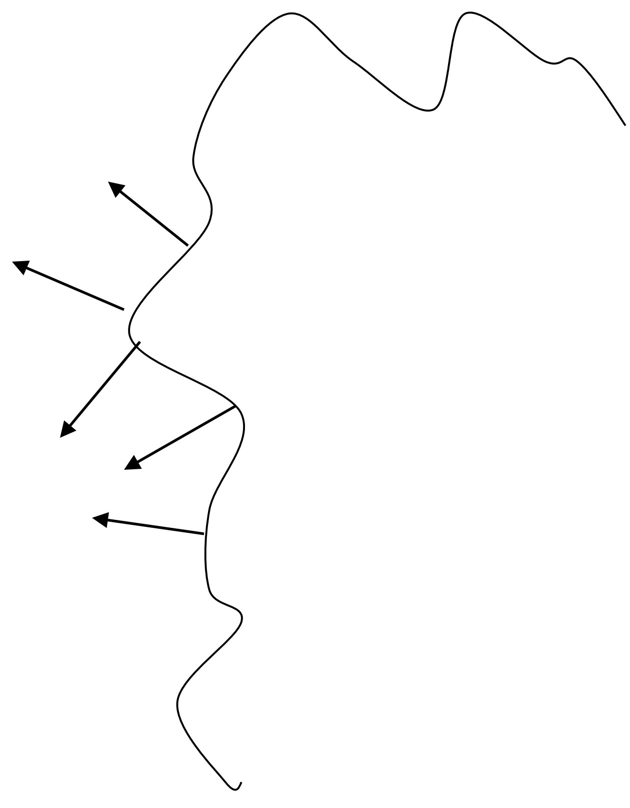
Bump Mapping Example



Crawfis 1991

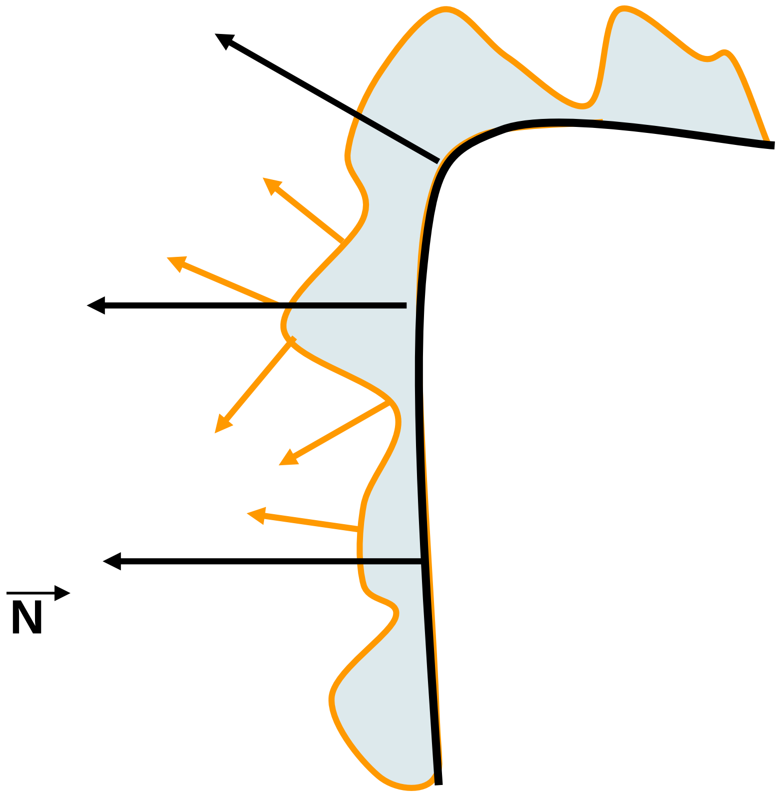
Bump Mapping

- Consider the lighting for a modeled surface.



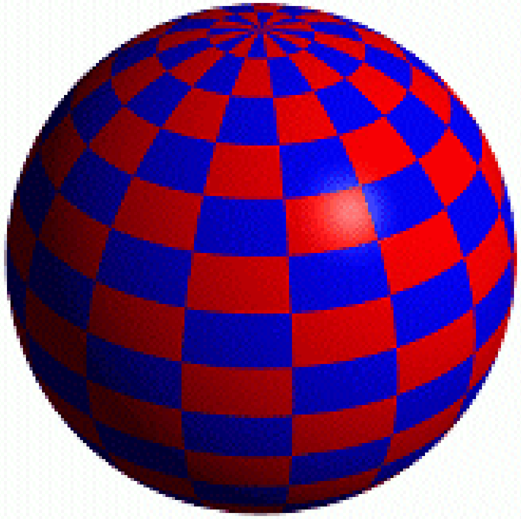
Bump Mapping

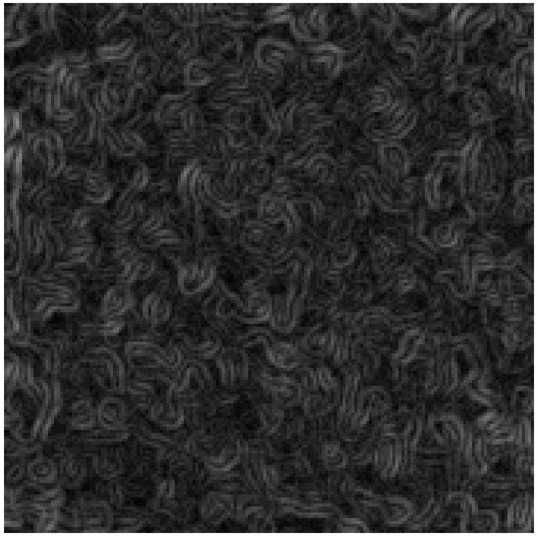
- We can model this as deviations from some base surface.
- The question is then how these deviations change the lighting.

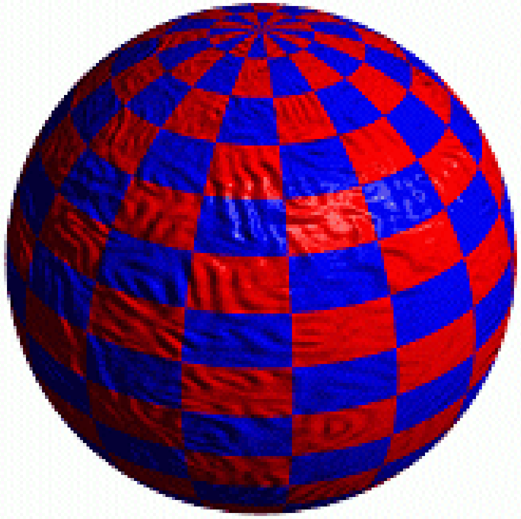


Bump Mapping

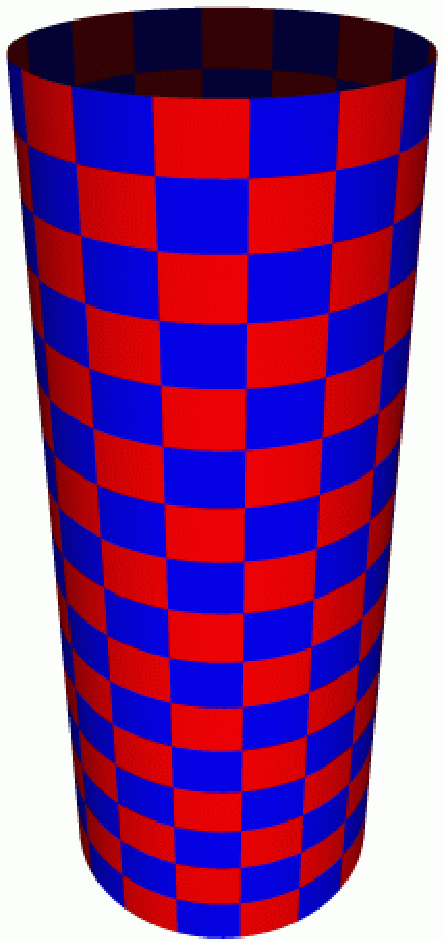
- Store in a texture and use textures to alter the surface normal
  - Does not change the shape of the surface
  - Just shaded as if it were a different shape


Sphere w/Diffuse Texture

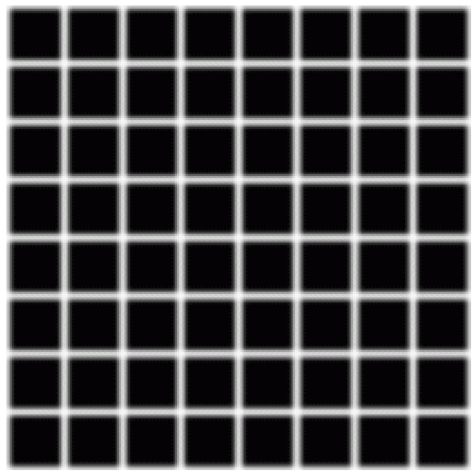

Swirly Bump Map

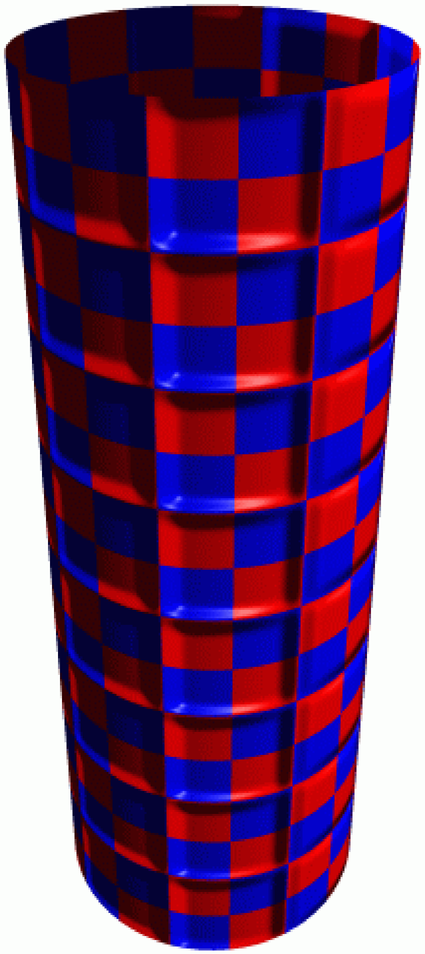

Sphere w/Diffuse Texture & Bump Map

Simple textures work great



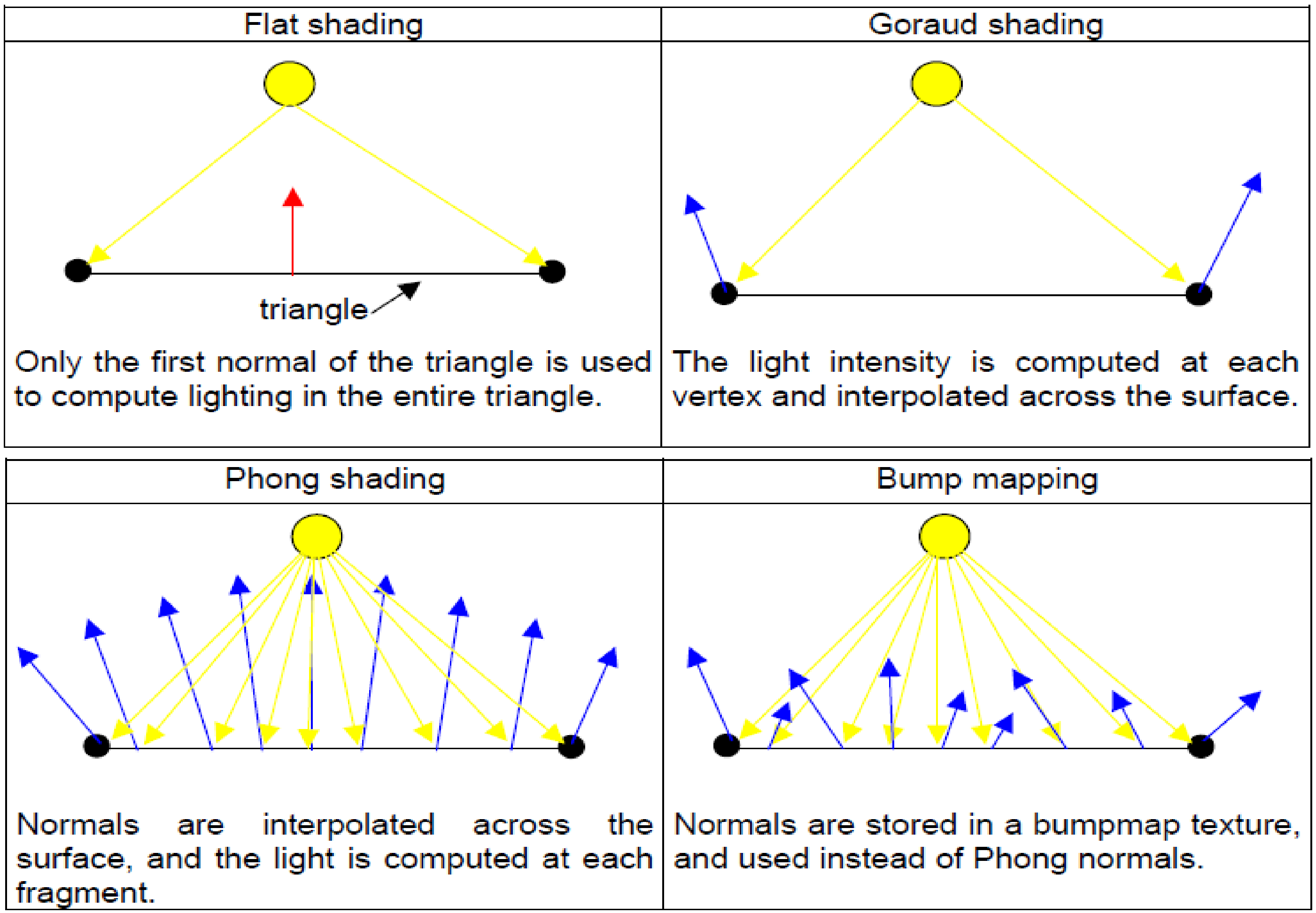
Cylinder w/Diffuse Texture Map





Cylinder w/Texture Map & Bump Map

Normal Mapping



Normal Mapping

Just texture mapped

Texture and normal maps

Notice: The geometry is unchanged. There's the same number of vertices and triangles. This effect is entirely from the normal map.

Normal Maps

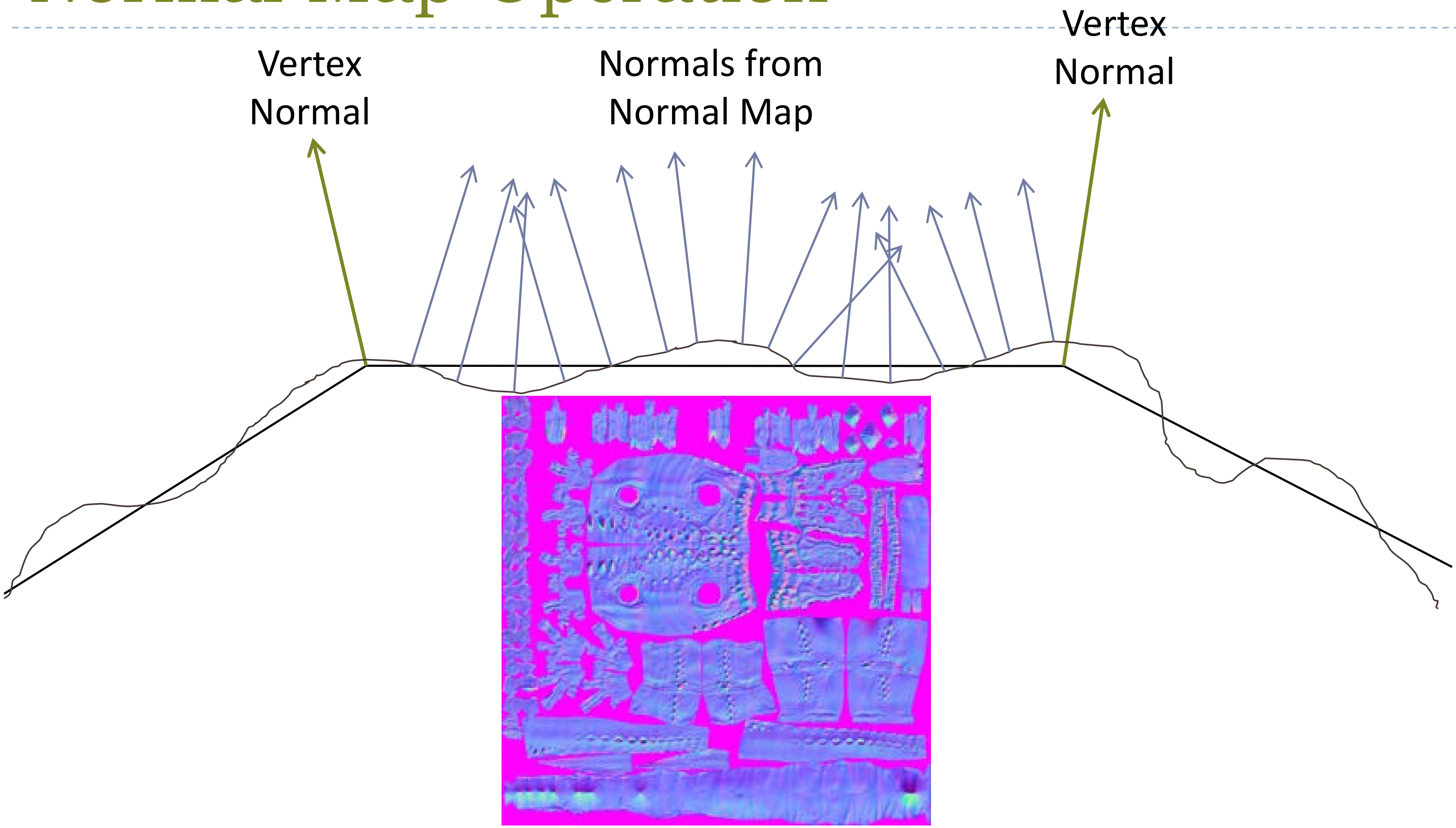
Diffuse Color Texture Map

Normal Map

Each pixel represents a normal vector relative to the surface at that point. -1 to 1 range is mapped to 0 to 1 for the texture so normals become colors.

→ Inverse of Normal Coloring

Normal Map Operation



For each pixel, determine the normal from a texture image. Use that to compute the color.

What's Missing?

- There are no bumps on the silhouette of a bump or normal-mapped object

→ Displacement Mapping