PG612:
Advanced Graphics Programming

Lecture 12:
Computational Shaders & GPGPU

André R. Brodtkorb <Andre.Brodtkorb@nith.no>

Outline

- Computational Shaders: Mandelbrot
- Repetition of parts of this course
  - Ask questions!
  - Will take more than two hours...
- Lab: Create your own Mandelbrot renderer

Today's Lecture

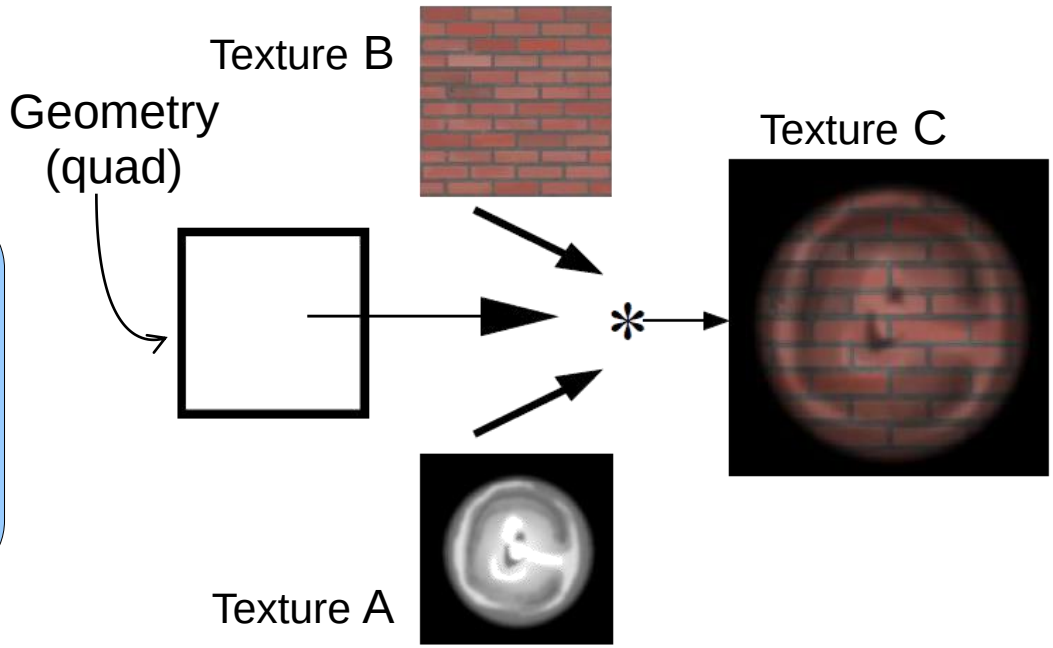
Introduction to GPGPU

- Our shades in this course have been for lighting and shadows (shading)
- GPGPU stands for “General Purpose computing on GPUs”
- The idea is to use the GPUs compute power for non-graphics applications
- Started around 2000 using fixed function hardware (up-to OpenGL 1.5)
- The programmable pipeline appeared, enabling more advanced GPGPU use.

Example: Multiplication

- Element-wise multiplication of many elements in parallel
- Render a full-screen quad to initialize computation
- Use fixed function to multiply
  - Blend mode
  - Multi-texturing

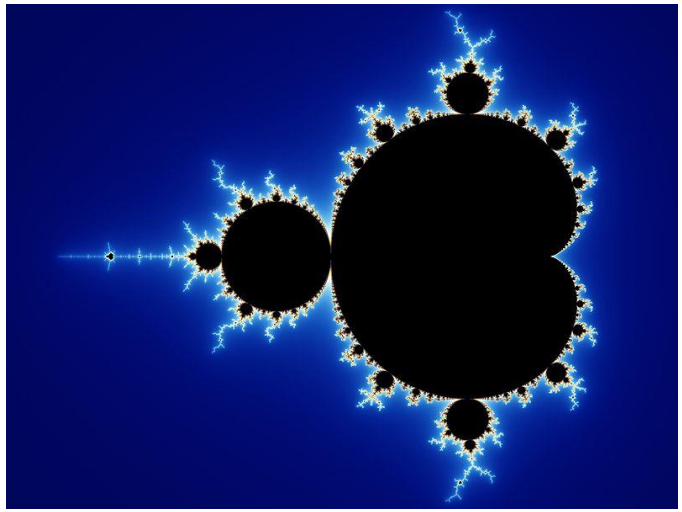
```
//Equivalent fragment shader:
void main() {
  float a = texture2D(a_tex, coord);
  float b = texture2D(b_tex, coord);

  gl_FragColor.r = a * b;
}
```



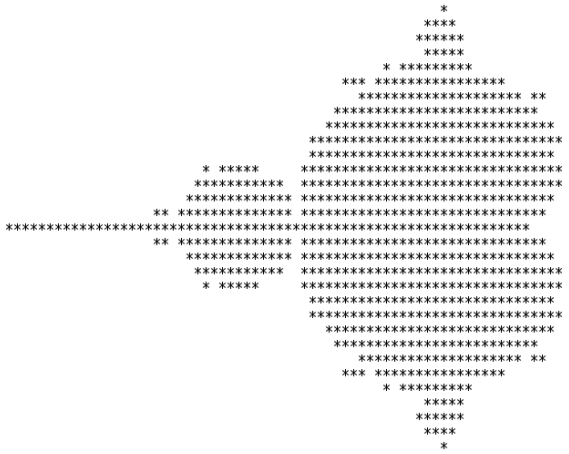
Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

The Mandelbrot Set

- The Mandelbrot set is a classical wallpaper-designer
- A fractal



http://www.youtube.com/watch?v=ZZl46mPUz8c



1978, first rendering (Matelski)

Rendering from Wikipedia, Wolfgang Beyer

The Mandelbrot Set

- Very simple definition: $P_c: z \rightarrow z^2 + c$
- "All complex numbers c, for which z does not tend towards infinity"
  - If |z| becomes larger than 2, we know it will tend to infinity.
- How to compute?
  - Use the pixel coordinate as the complex c
  - Set z_0 = 0, and compute z_n+1 = z_n^2 + c
  - If |z_k| > 2 for any k, abort
  - If k > k_max, abort and assume it is in the set.

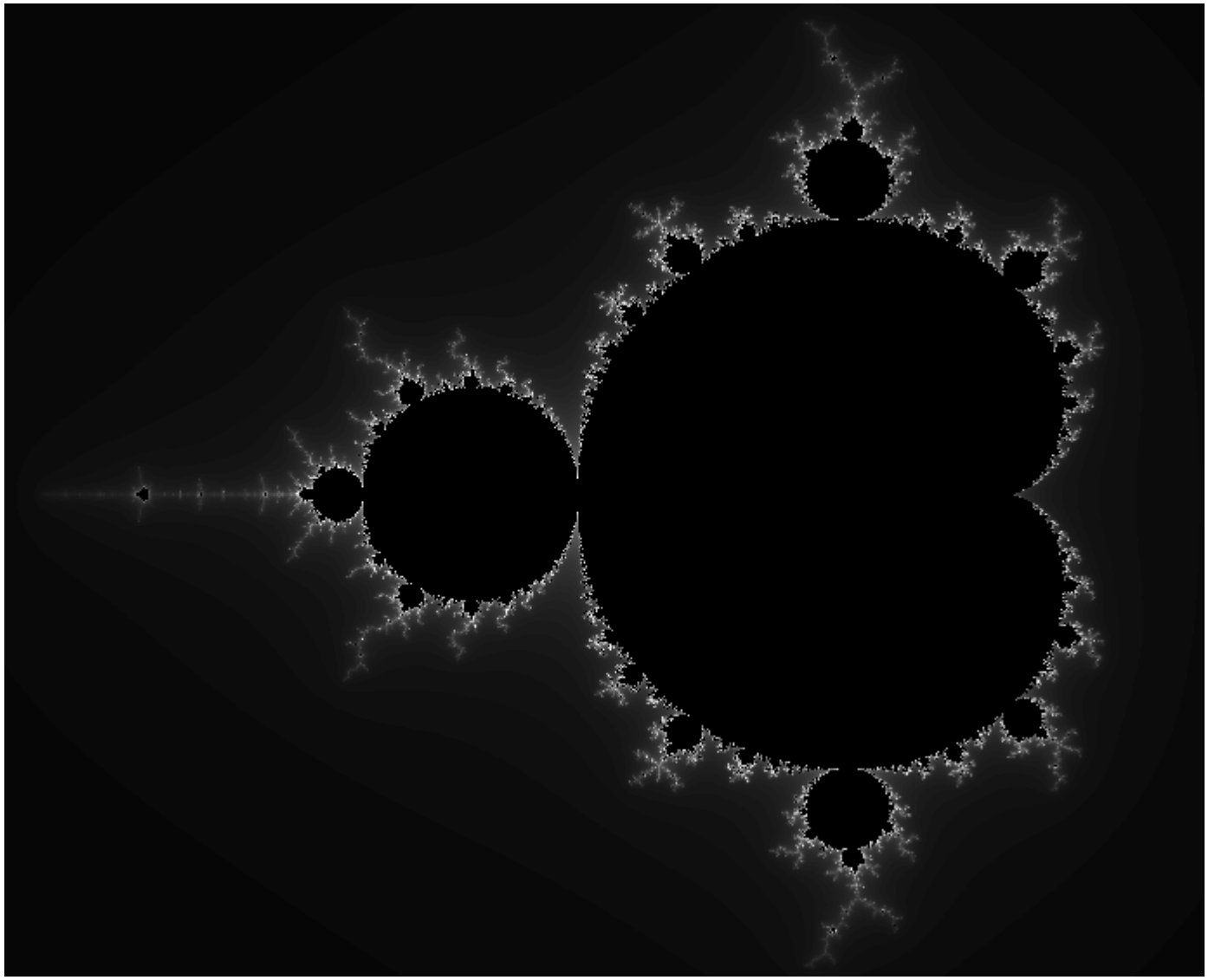
The Mandelbrot Set

```
//Initialize z to be c, the complex coordinate
complex z = c; //complex datatype

//Iterate until the length of z is larger than
//two, or until we have reached max_iterations
while (|z| < 2 && i < max_iterations) {
        //Update the value of z according to the
        //formula
        z = z^2 + c
        ++i;
}

if (|z| < 2) {
        out_color = 1; //Assumed to be part of the set
}
else {
        out_color = 0; //--”-- not to be part of the set
}
```

Coloring of the Mandelbrot Set

- The previous slide gives a black and white image: boring
- We can easily do better: use the color according to the number of iterations it took before escape:

```
if (i < iterations) {
        out_color = i / max_iterations;
}
else {
        out_color = 0;
}
```



Gray is dull!

- Gray is really really dull.... We want COLOR!
- Like in the movies we see online!
- But how to convert gray to color?

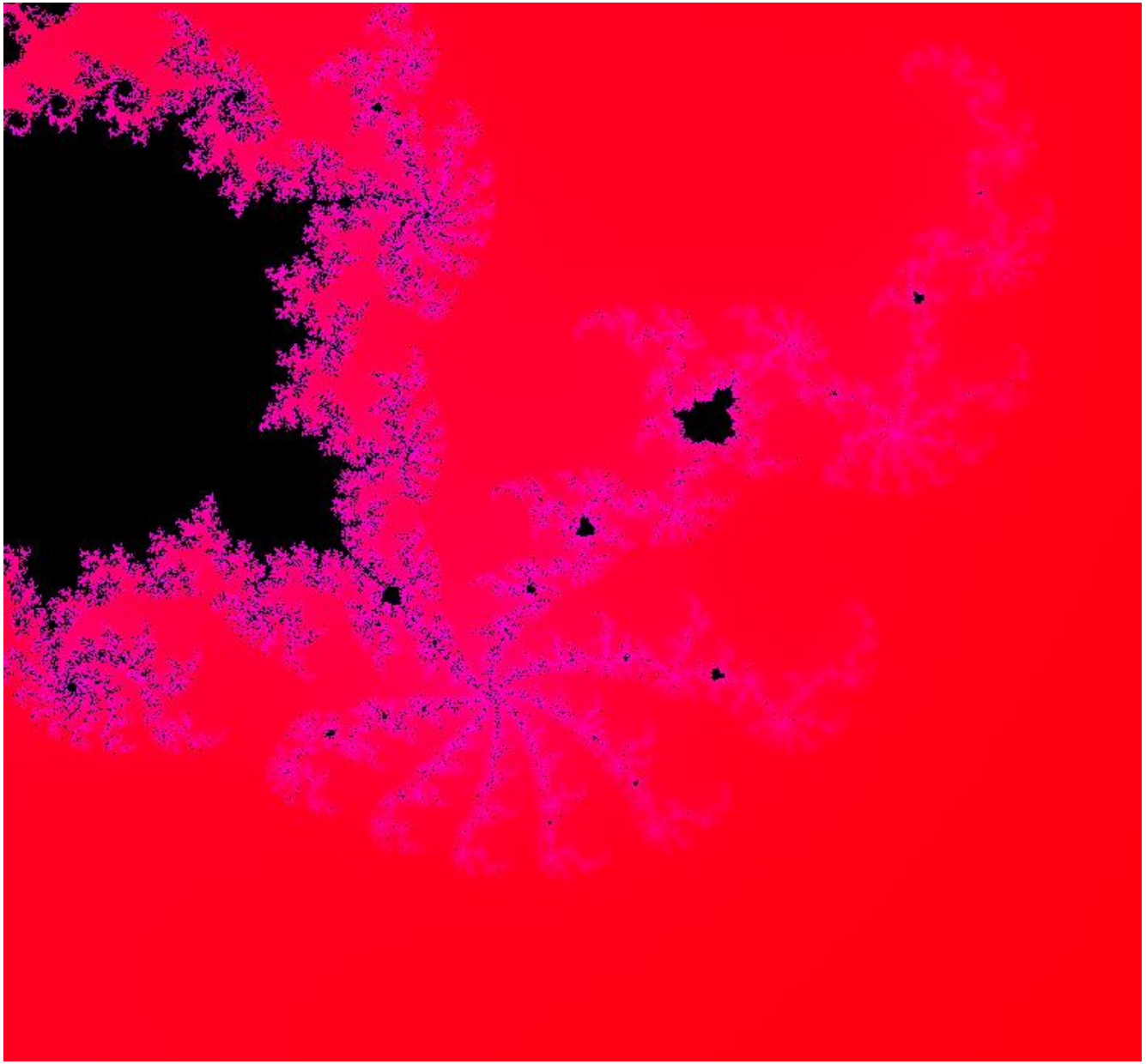
Linear Interpolation

- We can interpolate between colors: instead of using i/max_iterations as the color, we can use it as the parameter to interpolate between colors!

```
t = i / max_iterations;
```

- Remember linear interpolation?
  - x(t) = t*a + (1-t)*b

Interpolation from red (1, 0, 0) to violet (1, 0, 1) in RGB



Linear Interpolation

- Linear interpolation in the RGB color space gives ugly rainbow color cycles!

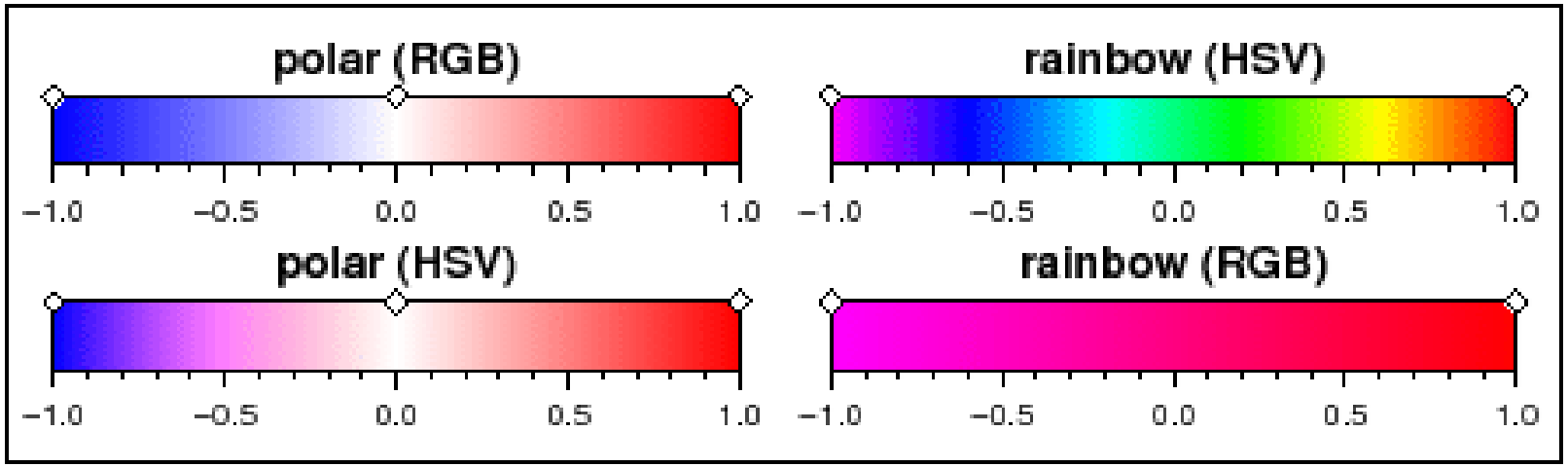


Image from
http://www.soest.hawaii.edu/gmt/gmt/doc/gmt/html/GMT_Docs/node214.html

HSV Interpolation

- Hue, Saturation, Value
- Hue: Degrees 0, to two pi
- Saturation: 0 to 1
- Value: 0 to 1
- Gives much better interpolation!

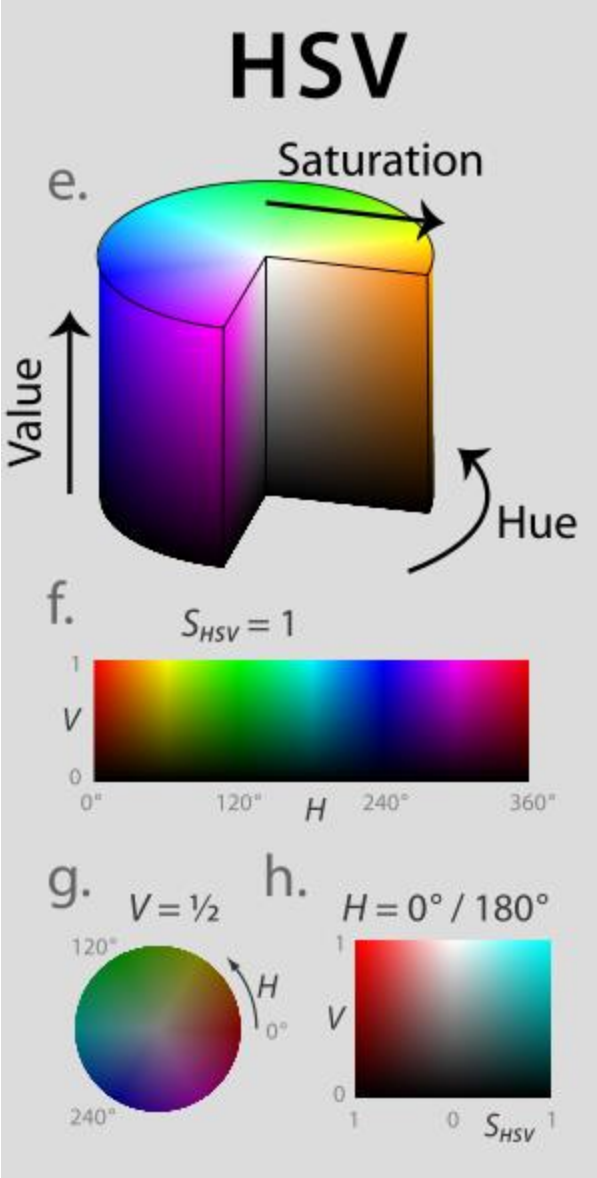
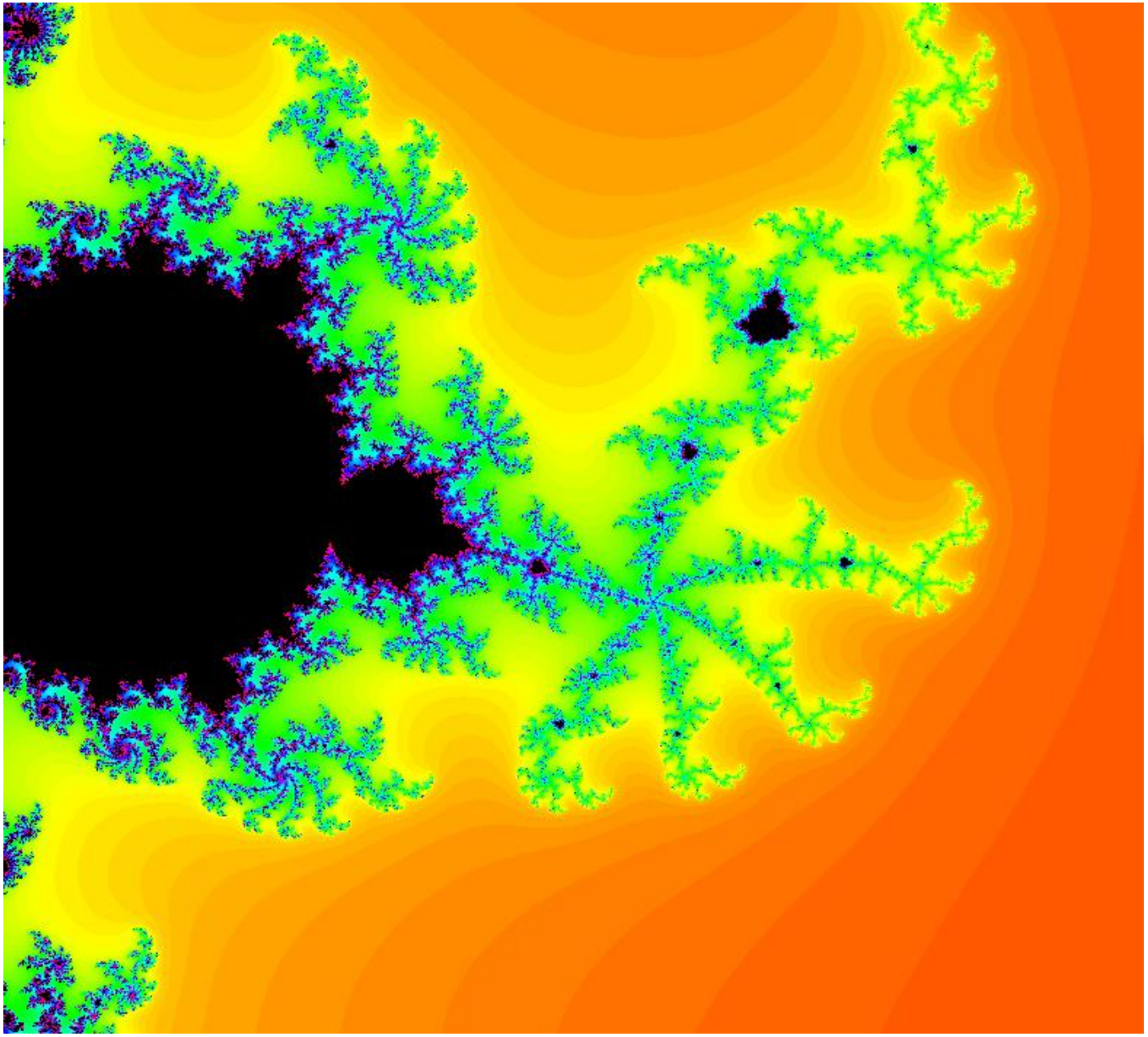


Image from Wikipedia, user Jacob Rus

HSV to RGB

- First, interpolate using the HSV color space
- Then, convert the resulting HSV color to the corresponding RGB color
- Write out the RGB color to screen

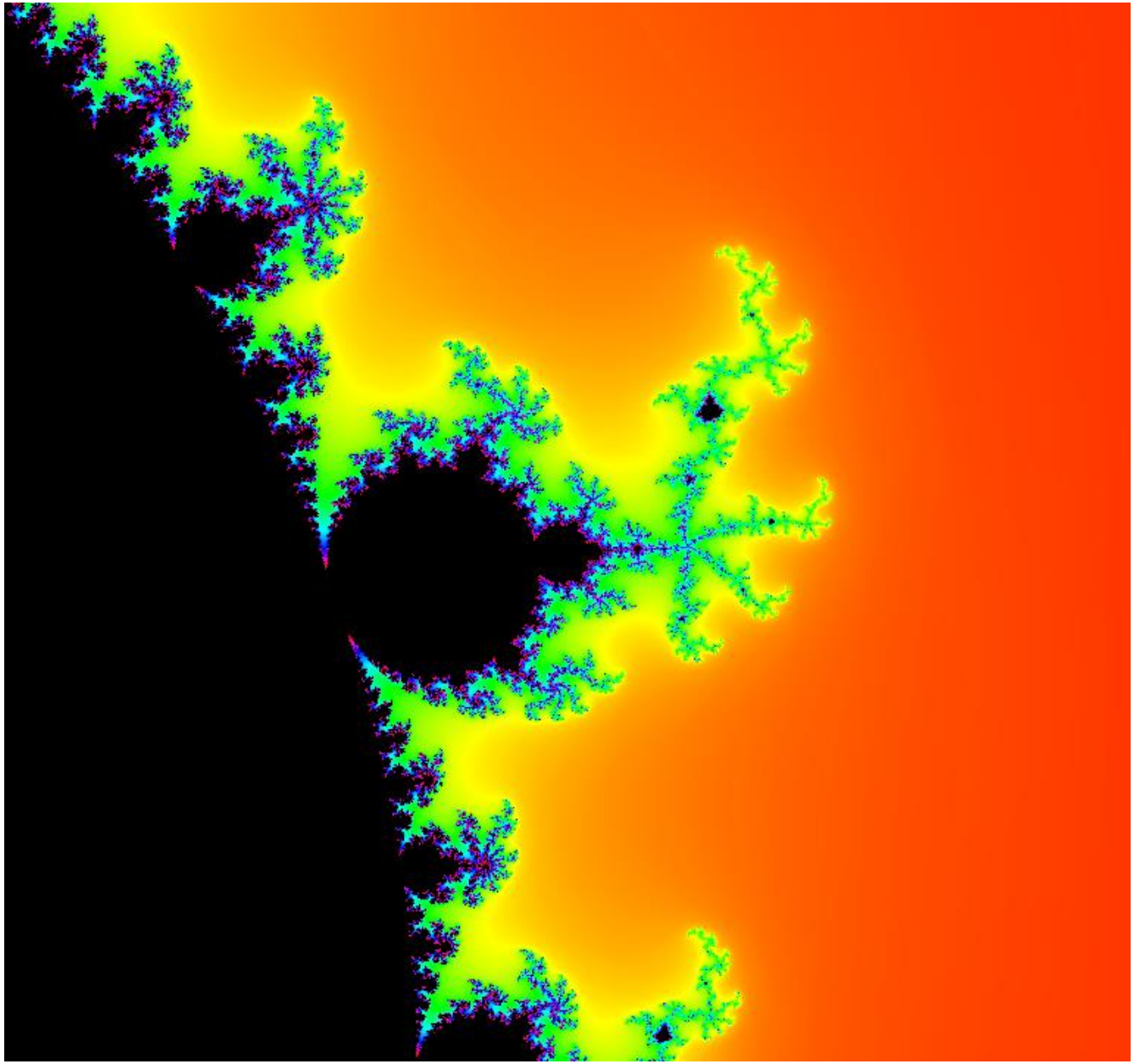
Interpolation from red (0, 1, 1) to red (2*pi, 1, 1) in HSV



Removing the banding...

- The banding is not very nice to look at... lets remove it
- We can use some different properties of the set and its convergence to get a continuous shading almost for free!

```
t = (i - log(log(|z|)/log(2.0))/log(2.0)) / max_iterations;
```



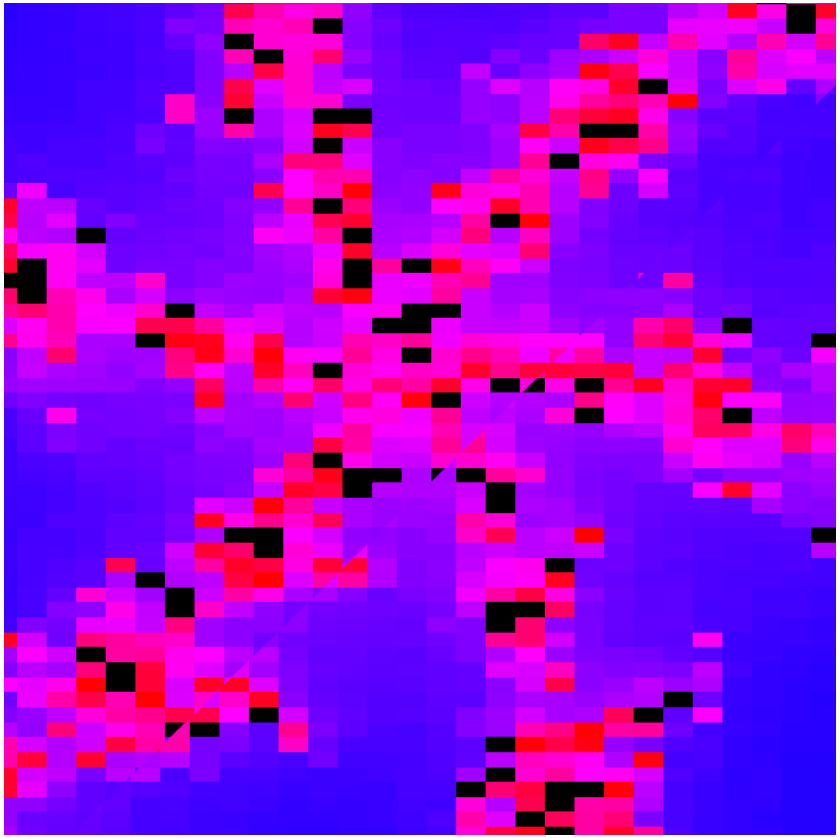
Still some banding

- We can increase the escape radius so that it is larger than two!
- This does nothing more than run more iterations before z becomes larger than the escape radius
- Essentially, we perform more iterations to get a smoother transition.

```
while (|z| < 5 && i < max_iterations) { … }
...
t = (i - log(log(|z|)/log(5.0))/log(2.0)) / max_iterations;
...
```

Zoom errors?

- Why do we get pixels here? (this is not magnified...)
- How can we solve this?

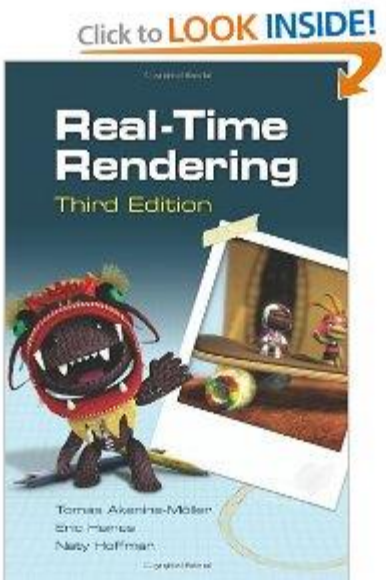


Repetition

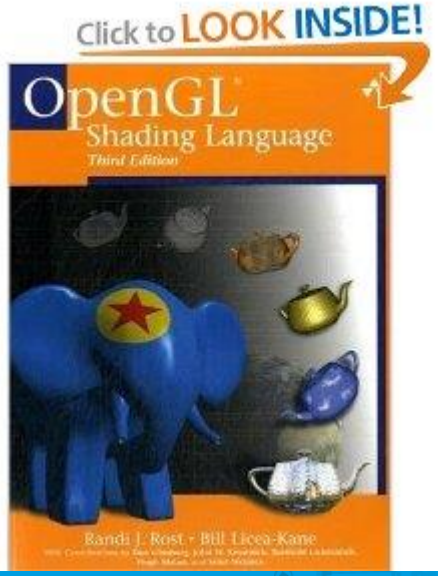
- We do not have time to repeat the whole course, so I will only go through a subset.

Books

- Real-Time Rendering, Third Edition
  - An incredibly good book.
  - I encourage you all to buy it and read it.
  - A «must» for all graphics geeks

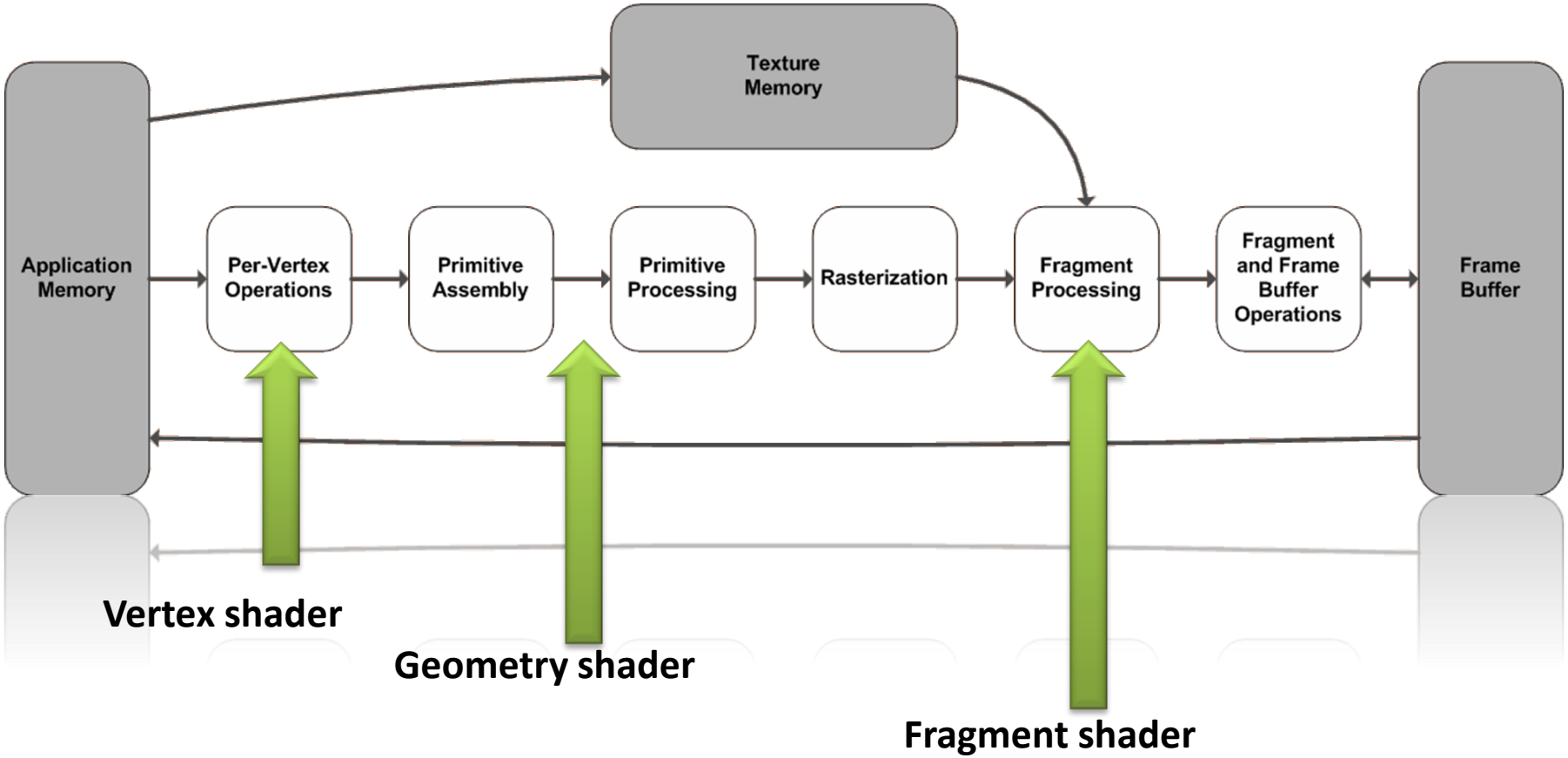


- OpenGL Shading Language (3rd Edition)
  - The orange book
  - A thorough reference for the OpenGL shading language
  - Don’t trust online tutorials: read this book and the Red book instead

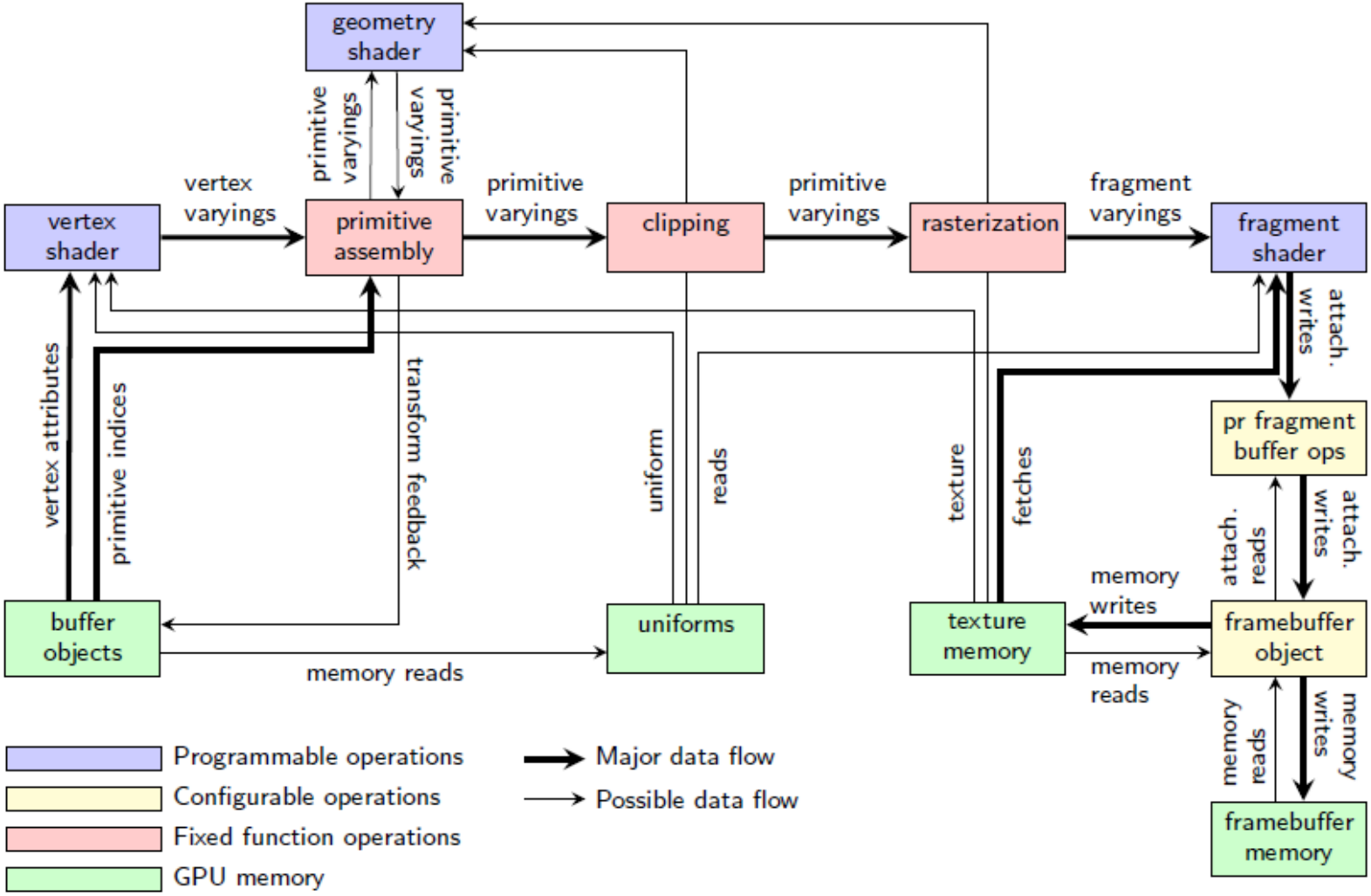


Programmable pipeline

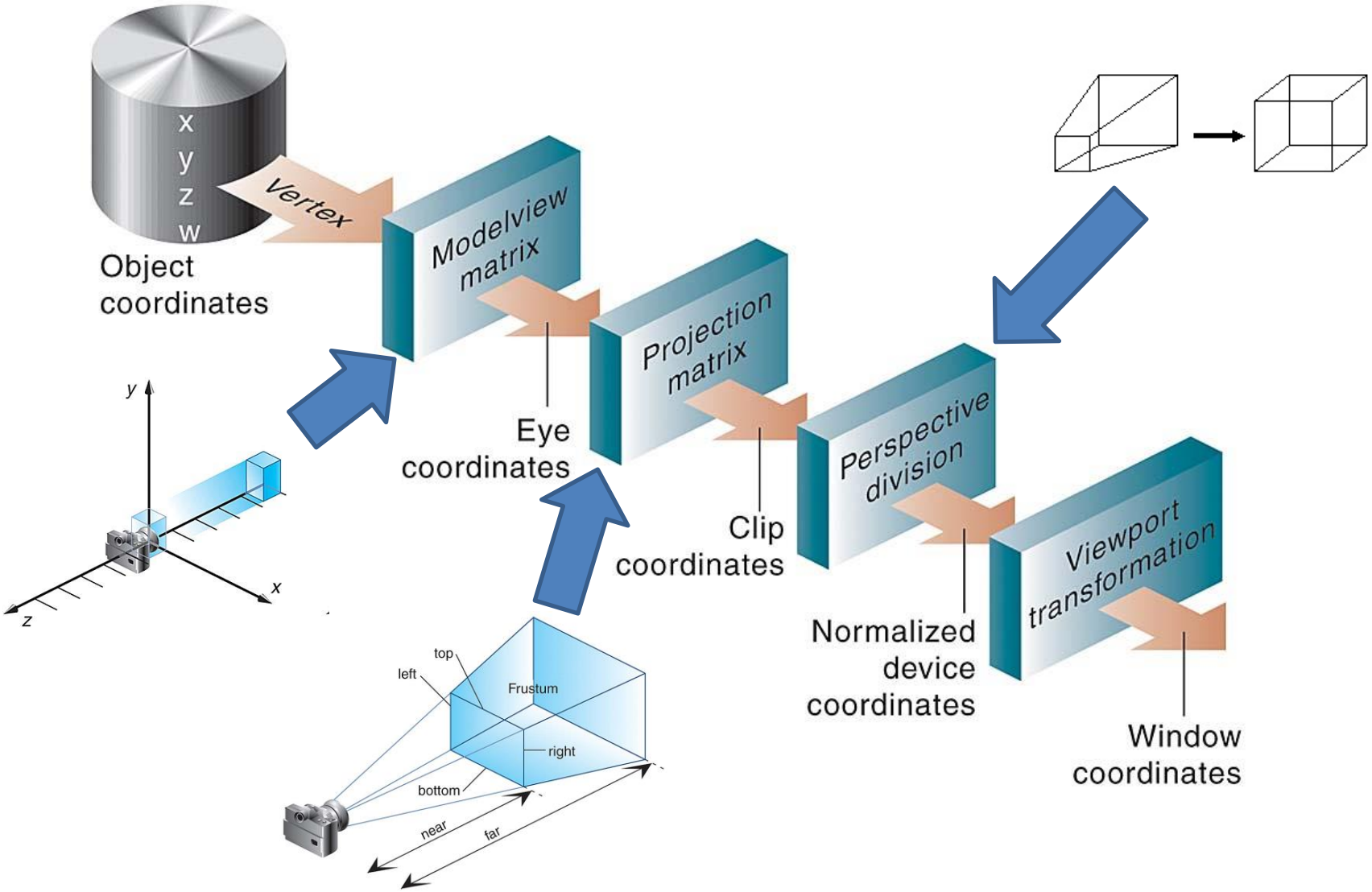
- Some steps in the pipeline become programmable



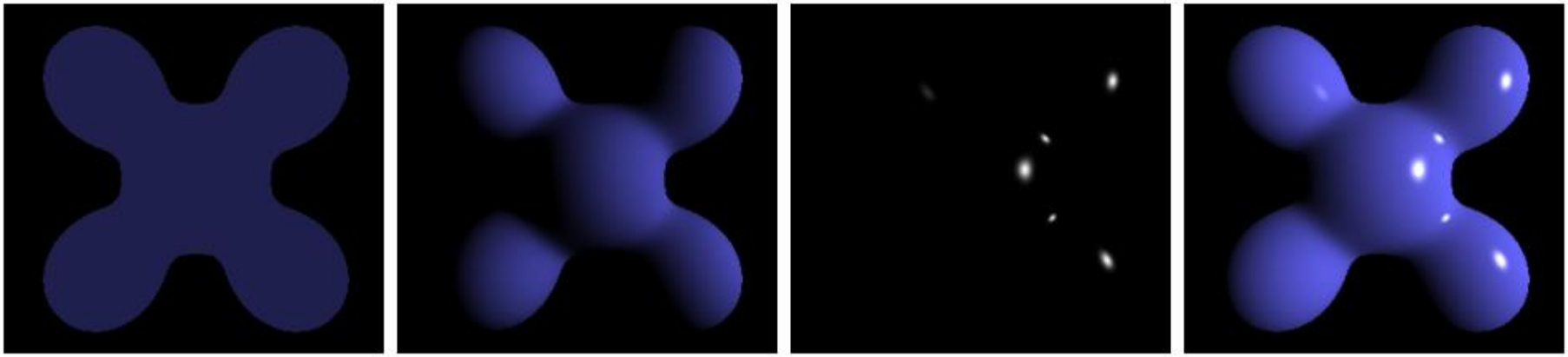
Modern OpenGL Pipeline



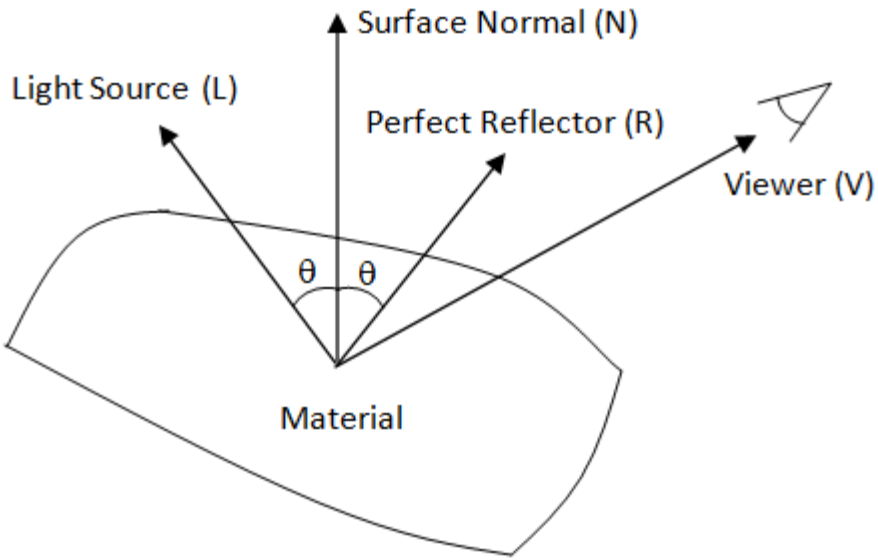
Spaces and transformations



OpenGL lighting equations

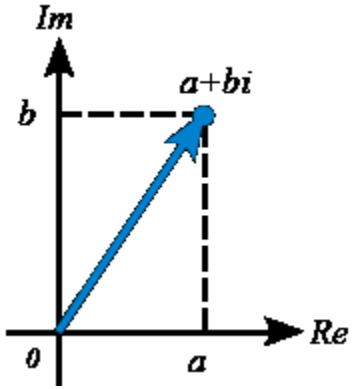


Ambient + Diffuse + Specular = Phong Reflection



Quaternions: "4D complex numbers"

- 2D complex number
  - Real and imaginary part



$$z = a + bi$$
$$i^2 = -1$$

- Quaternion

$$\hat{q} = iq_x + jq_y + kq_z + q_w$$
$$i^2 = j^2 = k^2 = -1$$

Mathematical background

- Definition:

$$\hat{q} = iq_x + jq_y + kq_z + q_w$$

$$i^2 = j^2 = k^2 = -1$$

$$jk = -kj = i$$

$$ki = -ik = j$$

$$ij = -ji = k$$

- Notation

$$\hat{q} = \left[q_x, q_y, q_z, q_w\right]^T = [\mathbf{q_v}, q_w]$$

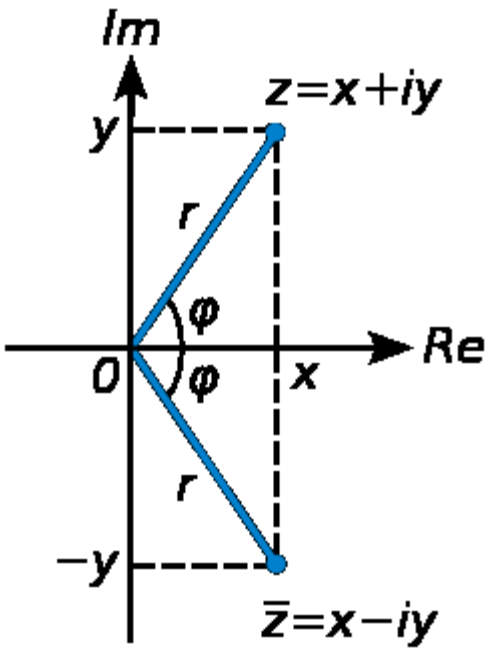
Unit Quaternions

- A unit quaternion has norm 1:
  - Norm:

$$n(\hat{q}) = \sqrt{\hat{q}\hat{q}^\circ} = \cdots$$
$$= \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$$

  - Conjugate:

$$\hat{q}^\circ = [\mathbf{q_v}, q_w]^\circ$$
$$= [-\mathbf{q_v}, q_w]$$



- Can represent rotations in 3D

Benefits of using quaternions

- Avoids Gimbal lock
- Concatenating rotations leads to growing errors
  - e.g., float a = 0.1; gives 0.100000001490116119384765625
- Matrices tend to become less and less orthogonal
  - Can be fixed using expensive orthogonalization
- Quaternions tend to become less and less unitary
  - Can be fixed using inexpensive normalization

Rendering with VBOs

- You can yourself specify VBO layout

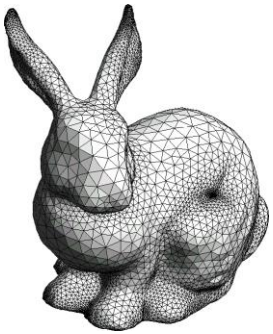
- Example: You have vertex coordinates, normals, and color:
  - One VBO for each: (VVVV) (NNNN) (CCCC)
  - One VBO for all: (VVVVNNNNCCCC)
  - Interleaved VBO: (VNCVNCVNCVNC)

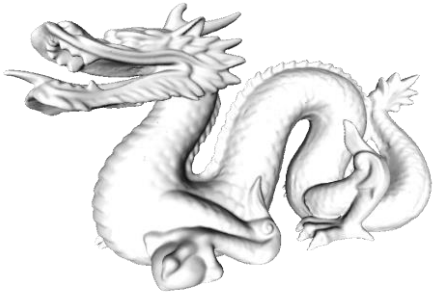
- Which will perform best?

More than one model per VBO?

- Lets take things further: use one VBO to hold several models
- Also remember glDrawElementsBaseVertex() for indexed rendering
  - glDrawElementsBaseVertex(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, &indices, 100)
  - [1, 2, 3, 4, 5] => [101, 102, 103, 104, 105]

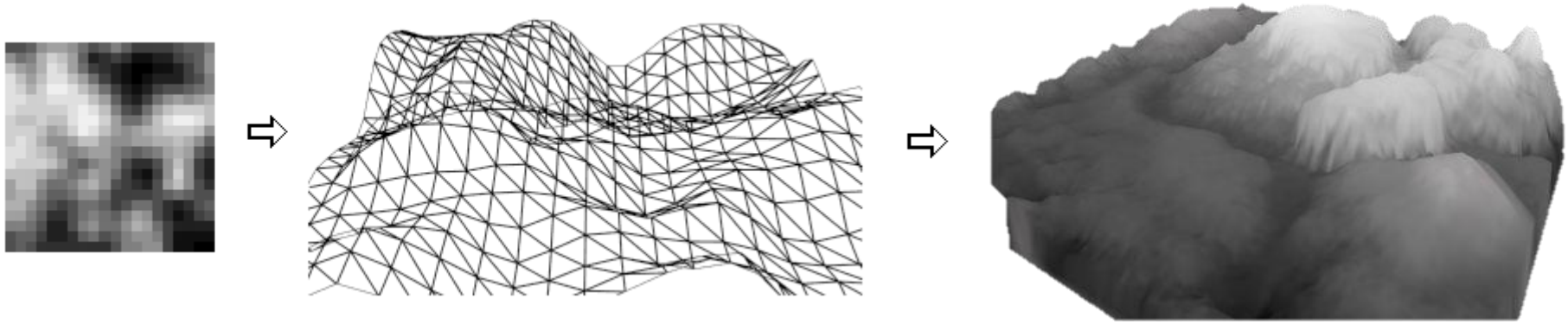
```
glDrawArrays(GL_TRIANGLES, 0, n_vertices_cow);
glDrawArrays(GL_TRIANGLES, n_vertices_cow, n_vertices_bunny);
glDrawArrays(GL_TRIANGLES, n_vertices_cow + n_vertices_bunny, n_vertices_dragon);
```





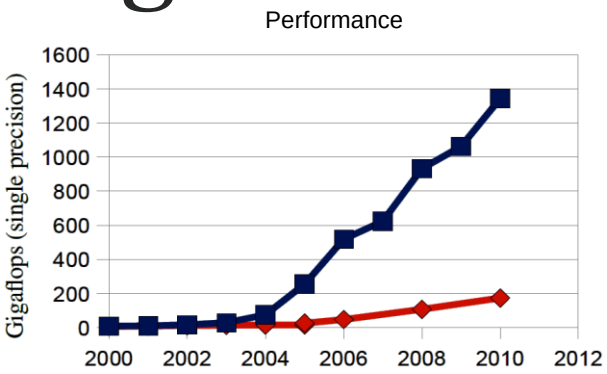
A Height map

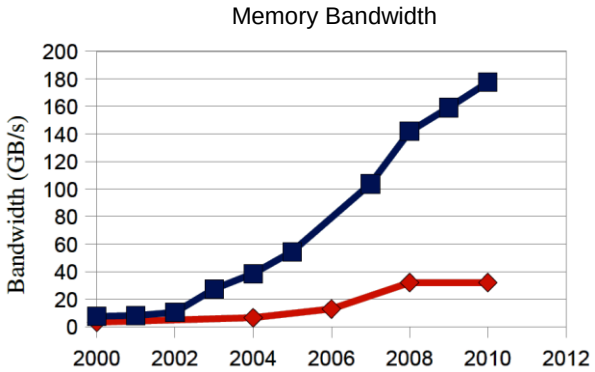
- Create a 2D texture
- Create a 2D mesh
- Displace vertices along the y-direction
  - Remember the standard camera orientation: y-axis is up, looking along negative z.
- Apply textures and shading



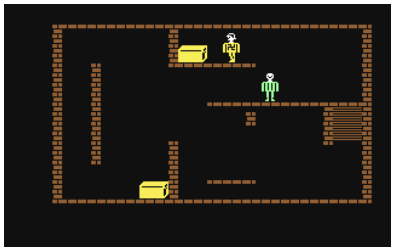
GPUs – Graphics Processing Unit

- Parallelism to the extreme
- Up-to 512 floating point units
- Tens of thousands of concurrent hardware threads





1981

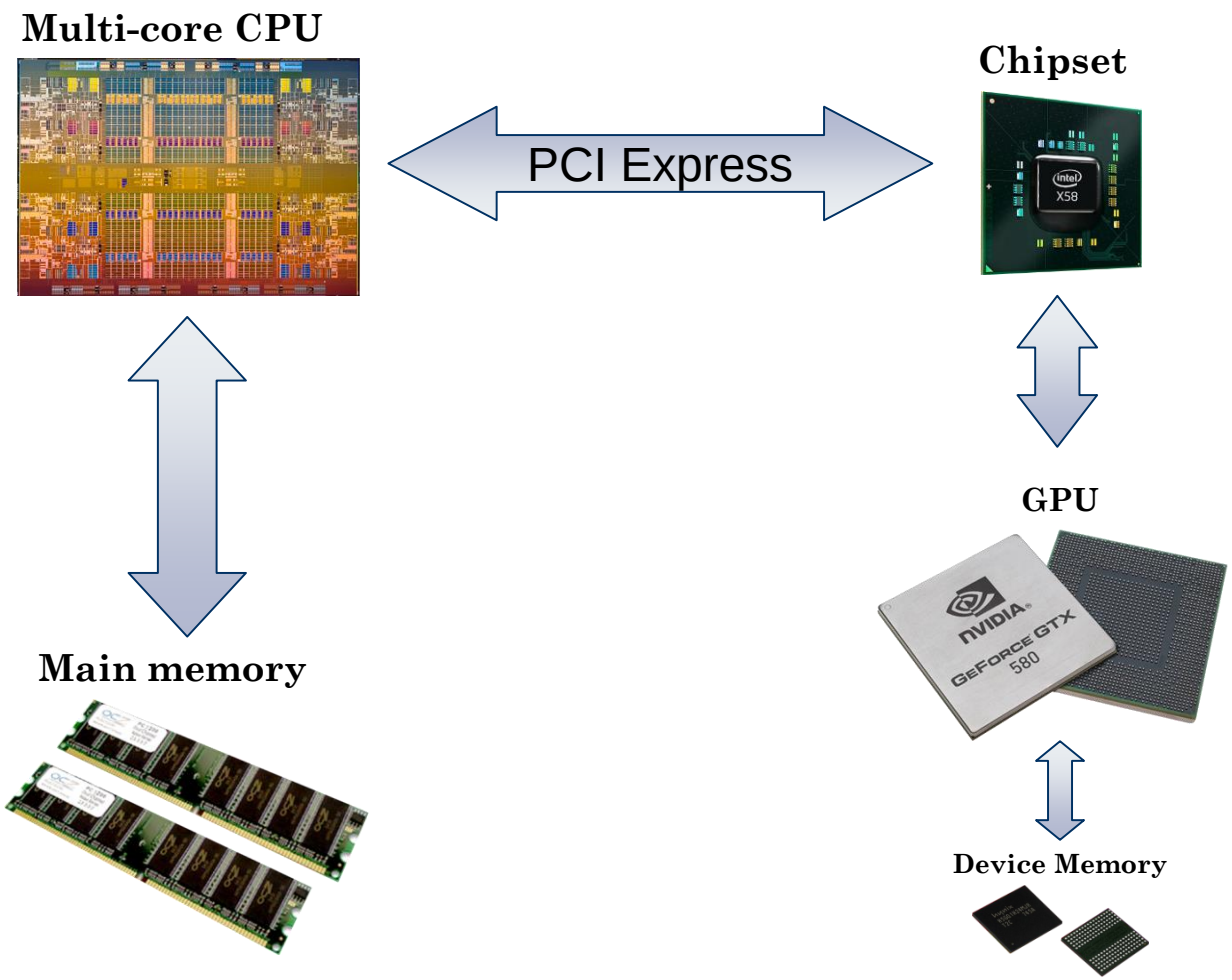


1992

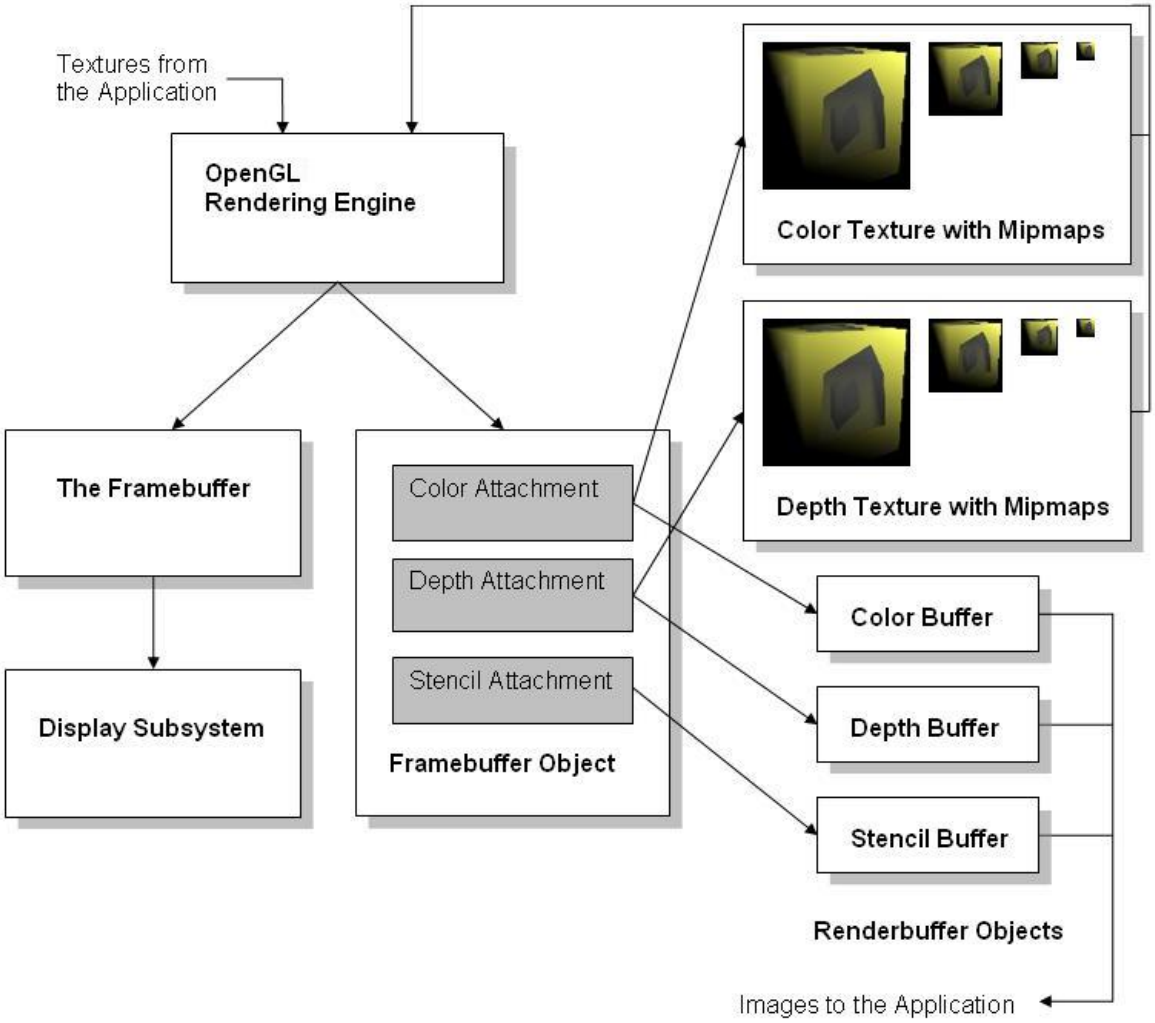
2001

2009

GPUs

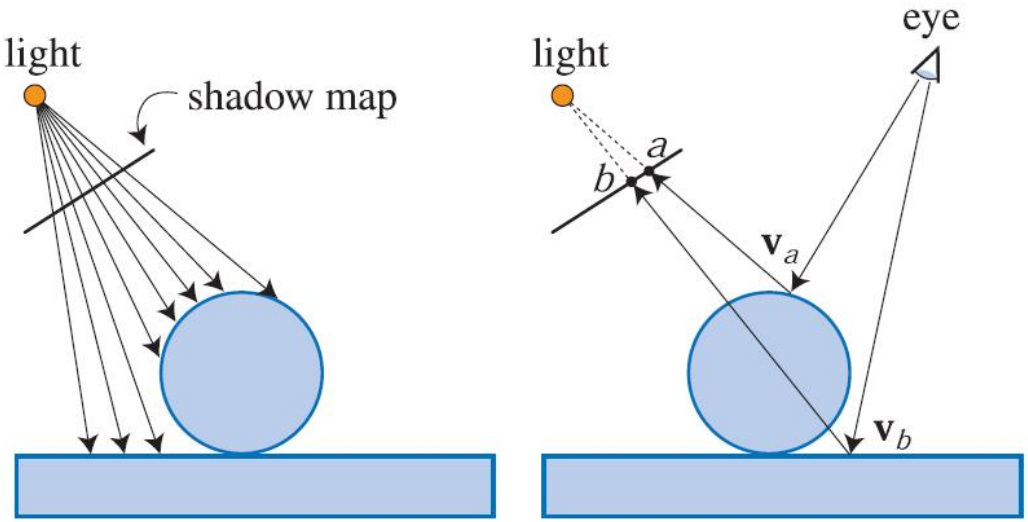

Multi-core CPU
PCI Express
Chipset
intel
X58
GPU
NVIDIA
GEFORCE GTX
580
Main memory
Device Memory

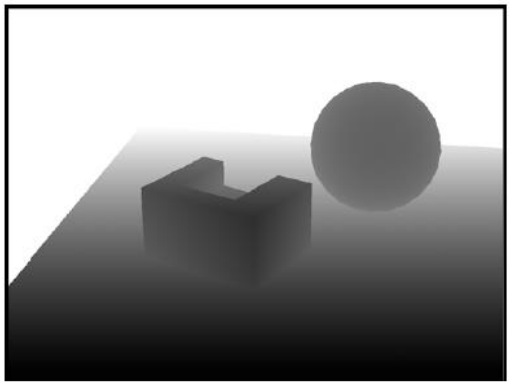
Summary FBO

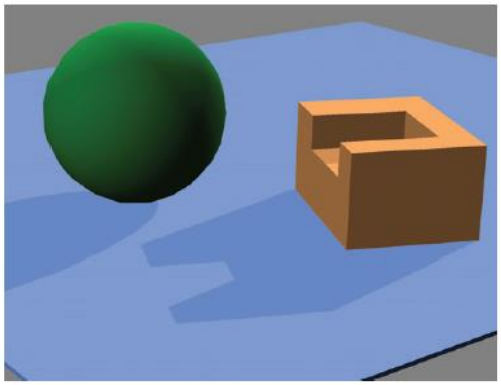

Textures from
the Application
OpenGL
Rendering Engine
Color Texture with Mipmaps
Depth Texture with Mipmaps
The Framebuffer
Color Attachment
Depth Attachment
Stencil Attachment
Framebuffer Object
Display Subsystem
Color Buffer
Depth Buffer
Stencil Buffer
Renderbuffer Objects
Images to the Application

Shadow Maps

- Texture based technique
  - Can be used for both soft and hard shadows
- Idea:
  - Render the scene as seen from the light
  - Store the depth value in a depth texture (d0)
  - Render the scene as seen from the camera
  - Compare the distance between the light and the pixel (d1) with the depth as seen from the light (d0)

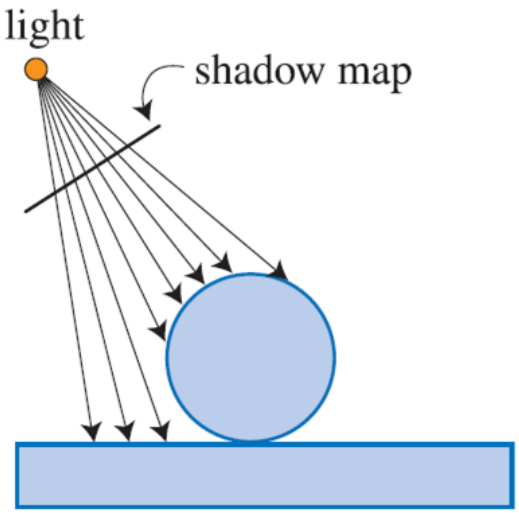




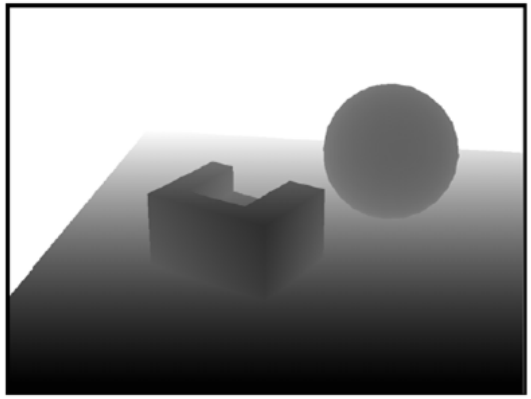


Rendering the Shadow Map

- Render the scene as seen from the light source to texture
  - Set up a light “view matrix”
  - Set up a light “perspective matrix”
  - Render to depth texture using a depth-FBO



- Debugging help: render a full-screen quad and texture with the depth texture

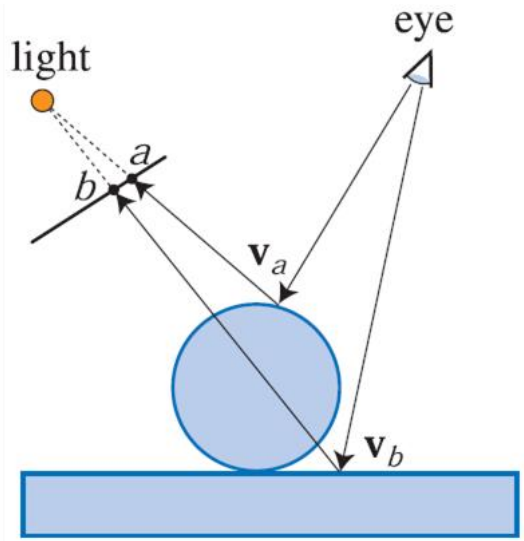


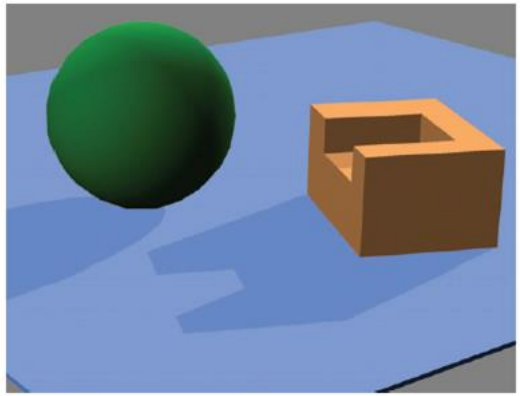
Rendering Shadows

- We have the depth as seen from the light
- Now, render the scene as seen from the camera, and compare depth values.

```
//Pseudocode vertex shader
vec3 pos_light = <light_transform>*position;
vec3 pos_cam = <cam_transform>*position;

//Pseudocode fragment shader
float depth_map_value = ???;

if (depth_map_value > pos_light.z) {
        //shadow
} else {
        //light
}
```

- Nice and simple,
  but how do we find depth_map_value?

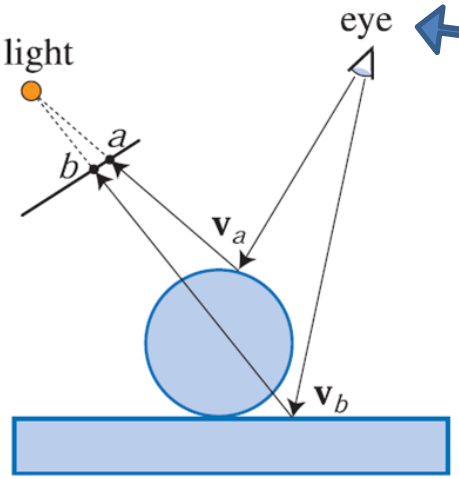




Repetition / Summary

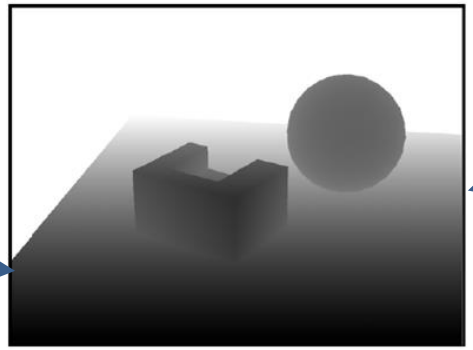
Rendering from light
(store in texture)

Object space
Model matrix
World space
Light matrix
Light Space
Light Perspective
Clip Space
Perspective divide
Norm device coordinates
Inverse T matrix
2D Texture Coordinates



Rendering from camera
(read from texture)

Object space
Model matrix
World space
Light matrix
Camera Space
Light Perspective
Clip Space
T matrix
3D Texture Coordinates

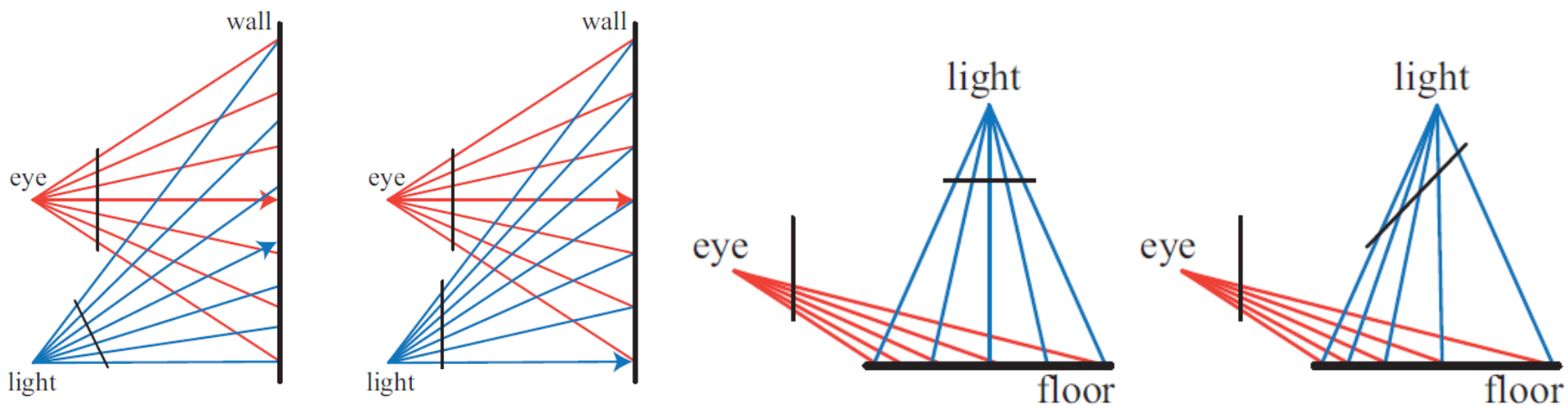


Store depth from light

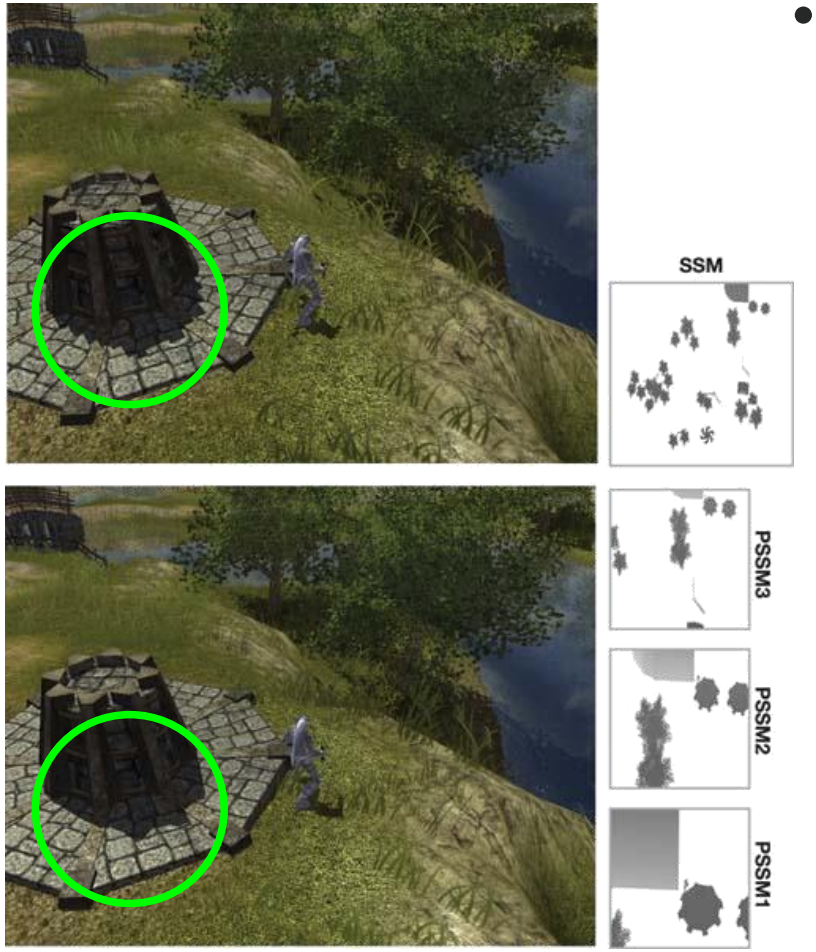
Read and compare depth values

Increasing Resolution

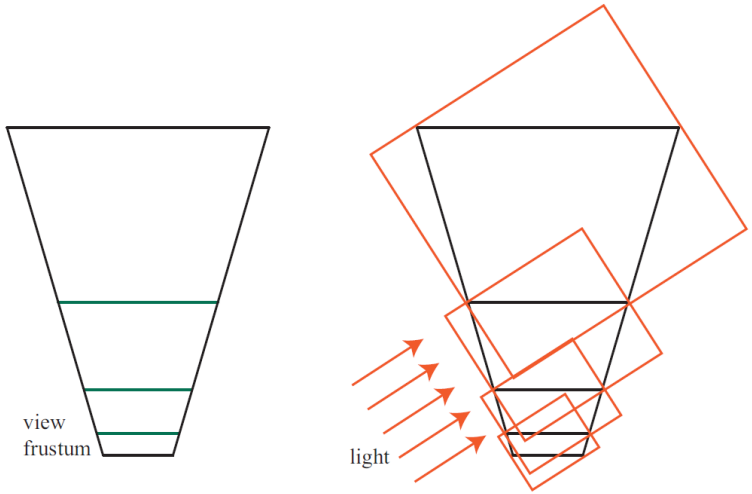
- We can alter the view frustum to increase resolution near the camera
  - Tilt the light view direction towards the viewing direction
  - Adjust the left/right/top/bottom planes of the frustum
  - This “tilts” the near and far planes, giving higher resolution



Parallel Split Shadow Mapping



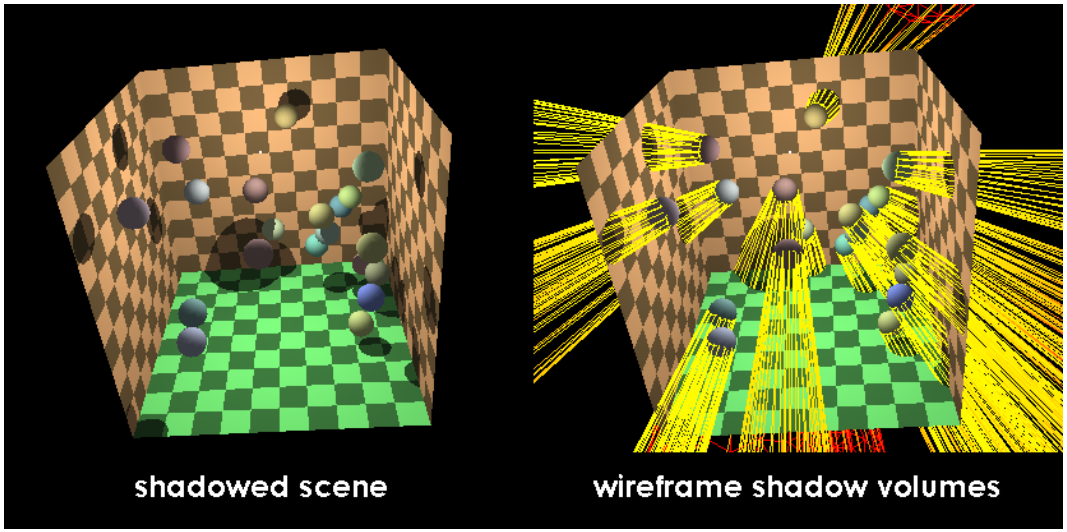
- Instead of using one large shadow map, use multiple smaller, each covering part of the view frustum



Shadow Volumes: Depth Pass

- First, find all silhouette edges, as seen from the light source
- Then, extend geometry from the silhouette edges away from the light source.

- Render the scene geometry
- Disable writing to depth and color buffers
- Render the shadow volume front faces and increment the stencil buffer for each front face
- Render the shadow volume back faces and decrement the stencil buffer for each back face
- Enable writing to color buffer
- Render the shadow volume back faces using the stencil test: shadow areas have more front faces than back faces.



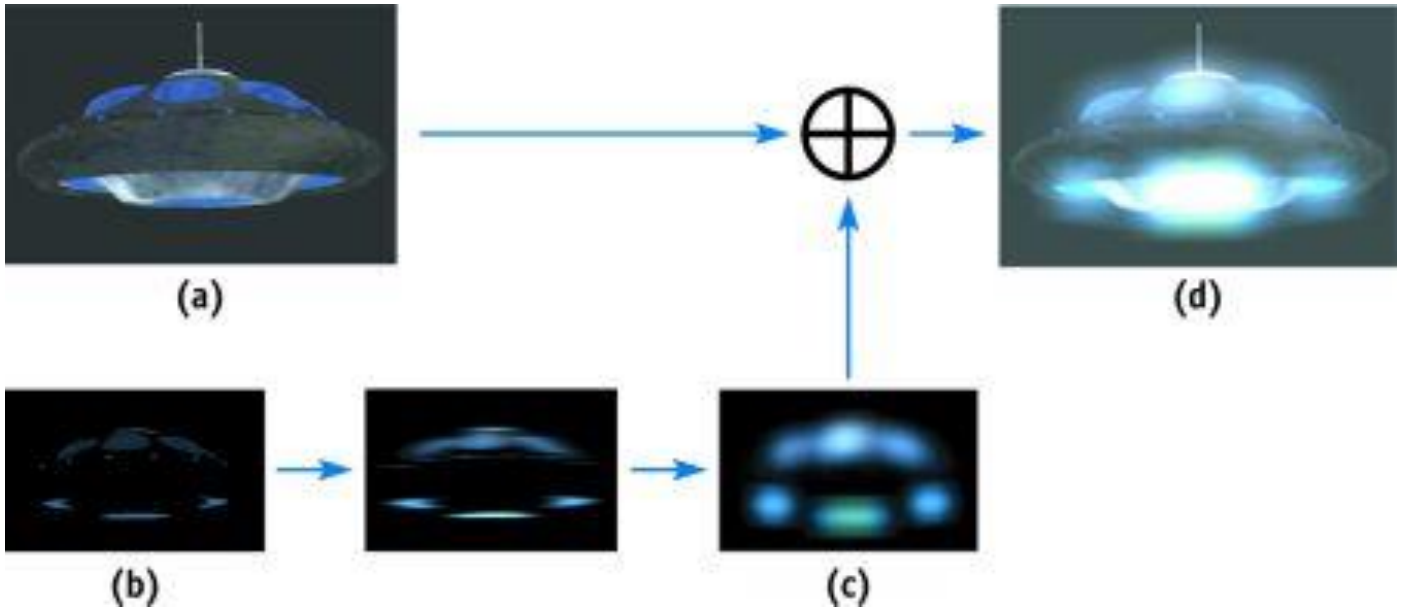
Multipass rendering

- We cannot always calculate everything in one shader pass:
  - E.g., Shadows

- It is not always a good idea to calculate everything in one shader pass
  - E.g., depth of field

Bloom in games

- Render scene to an FBO (“scene texture”)
- (Locate bright areas)
- Downsample and blur “scene” (“bloom texture”)
- Add the bloom to the scene texture, and render to screen



HDR Rendering

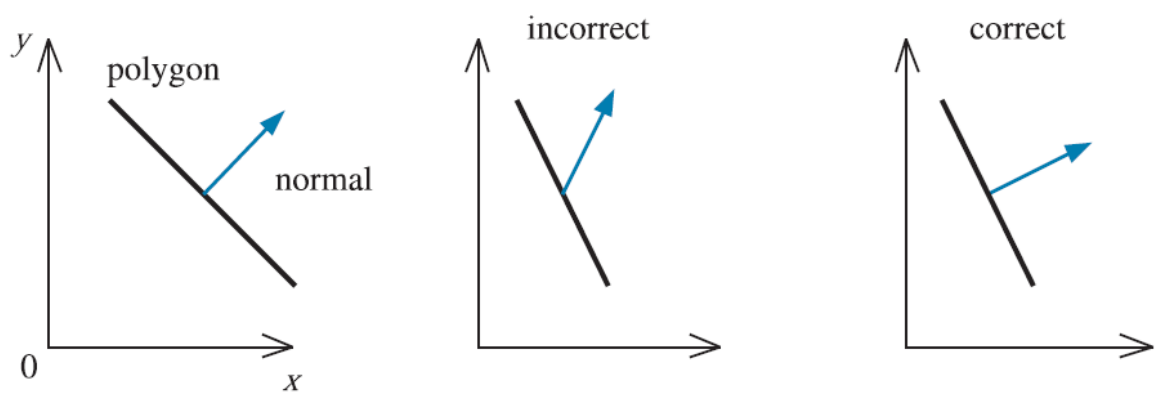
- How can we use a computer monitor to show e.g., 500 to 25000 lux in the same scene?
- Computer monitors have low dynamic range
  - Colors are clipped to [0, 1]
- Our eyes adjust to different light intensities automatically
- The aim of high dynamic range is to show details in both light and dark areas of a scene

HDR Rendering / Photography

Transformation of Normals

- We know np=0, and want to find n'p'=0
- Insert identity matrix
- Use that M^-1M = I
- Mp = p', which gives us n' = nM^-1
- Reordering gives us $n' = M^{-1^T} n$

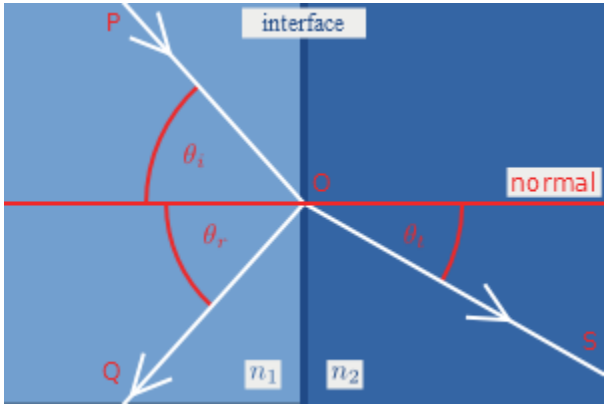
$$n \cdot p = 0$$
$$nIp = 0$$
$$nM^{-1}Mp = 0$$
$$n' \cdot p' = 0$$



scaled by 0.5 along the *x* dimension

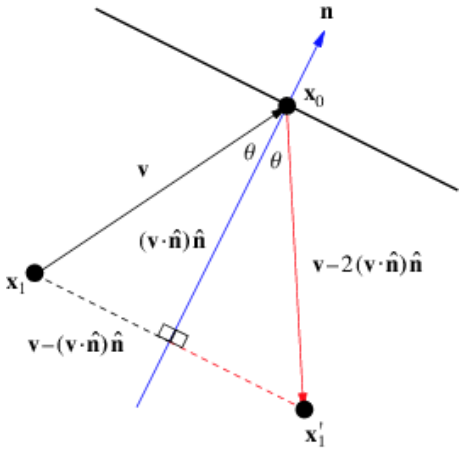
Fresnel Equations

- We have an incoming ray, P, going from a medium with refraction index n1 (e.g., air) to a medium with refraction index n2 (e.g., water)



- We can find the angle of refraction by using Snells law $n_1 \sin\theta_i = n_2 \sin\theta_t$

- The relfected ray is easily found using the dot product



Schlick's approximation

- Now that we have computed Q and S from P and the normal, we need to find the amount of reflection and refraction.

- Instead of computing the true ratio of reflectance (which can be quite expensive), we can use Schlick's approximation

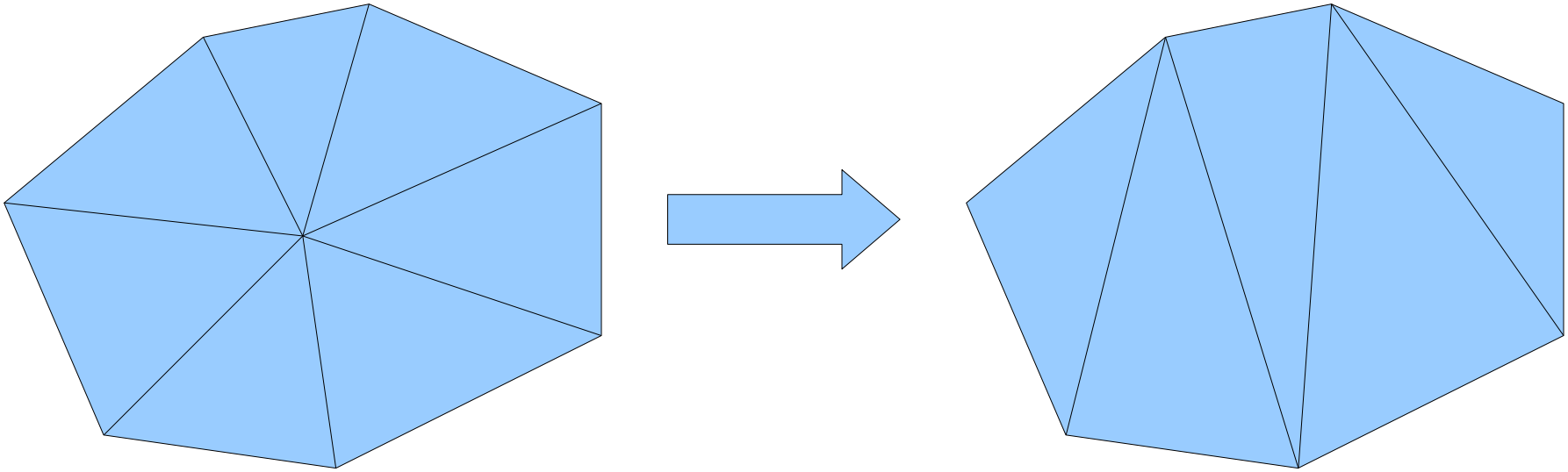
$$R_F(\theta_i) \approx R_F(0^o) + \left(1 - R_F(0^o)\right)(1 - \cos\theta_i)^5$$

- RF(0) is computed from the refractive index for the n1-n2 pair of materials

$$R_F(0^o) = \left(\frac{n_t - n_i}{n_t + n_i}\right)^2$$

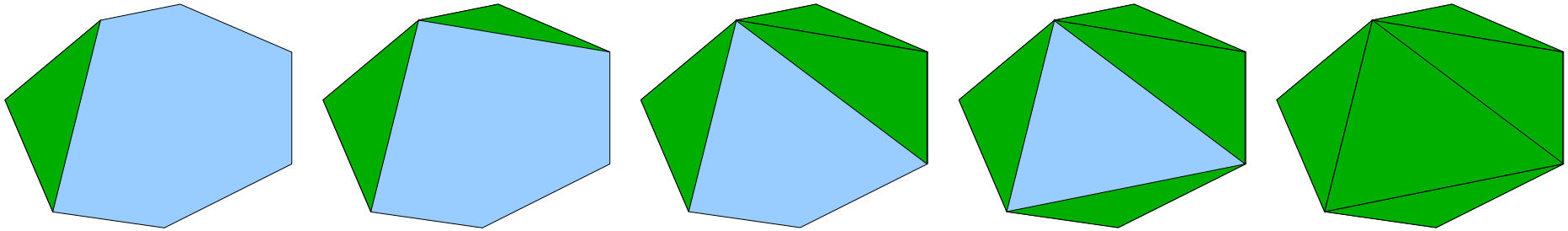
Vertex Removal

- A basic way to perform simplification is to remove a vertex, and retriangulate the hole



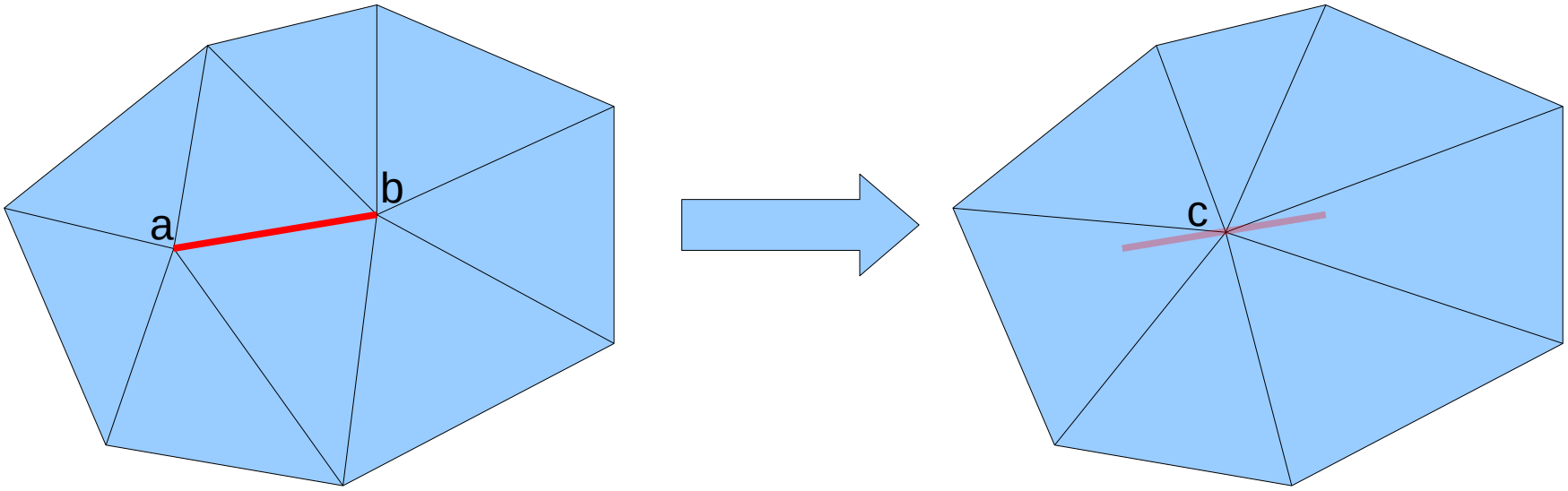
Ear Clipping

- Simple algorithm: “ear clipping”
- Find an “extruding ear” (randomly, or choose an error metric), and clip it of as a triangle
- Continue until there is only one triangle left



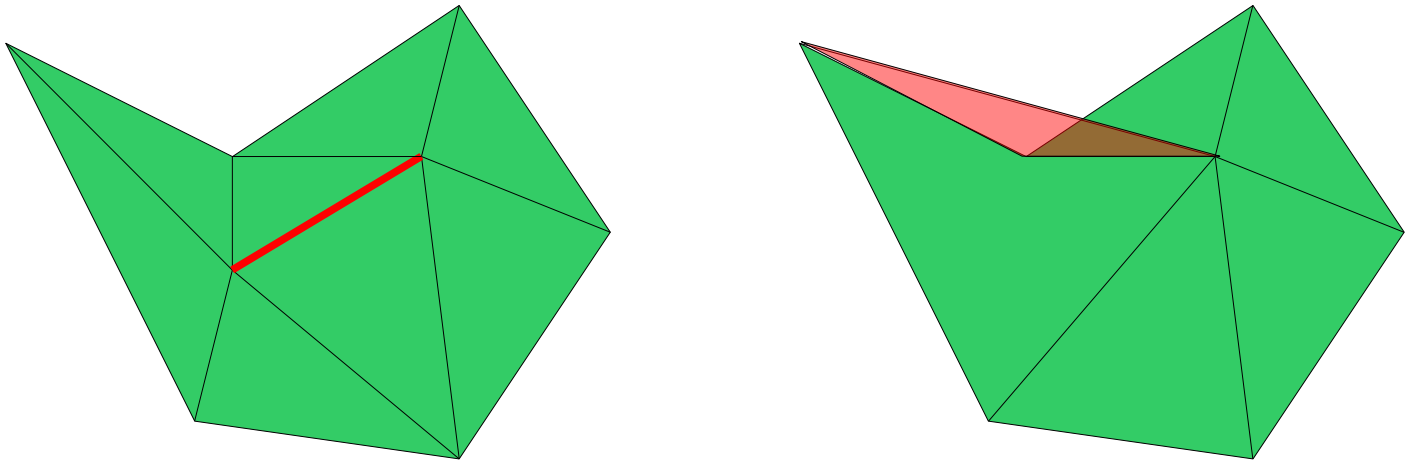
Edge collapse

- Instead of removing a vertex, we can also remove an edge
- Placement the new vertex, c, (which replaces a and b):
  - Halfway between a and b
  - Minimize some error
  - Etc.



Pitfall #1: Flipped triangles

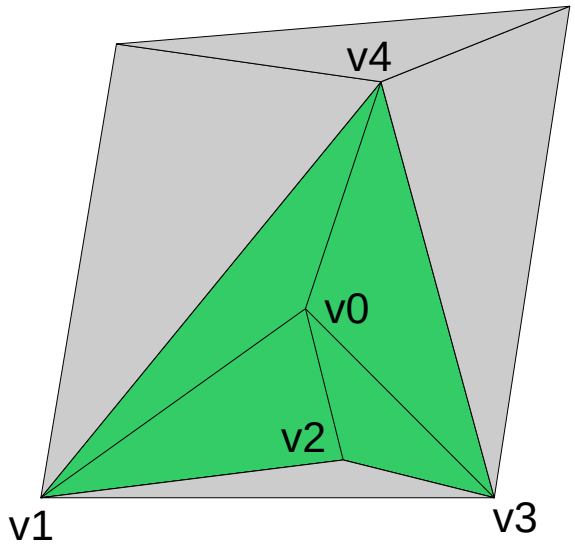
- Flipped triangles
  - Illegal in 2D, visual artifacts in 3D (culling++)

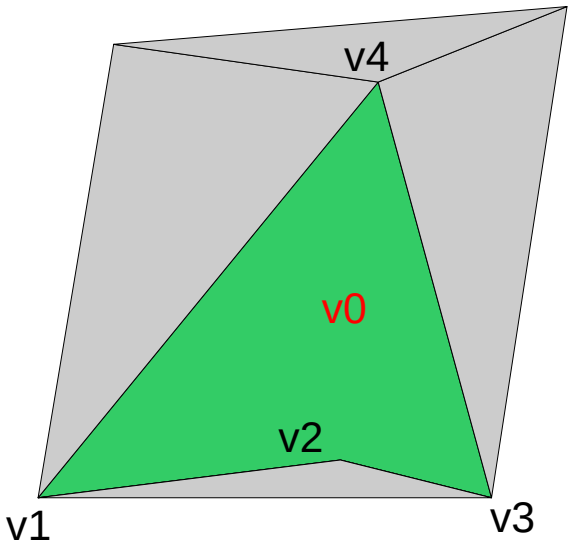


- Can be somewhat detected by checking normals in 3D
  - But difficult to perform robustly

Pitfall #2: Degenerate triangles

- Remove v0 and retriangulate
- Retriangulating by creating an edge between v1 and v3 is illegal: They already form an edge in another triangle

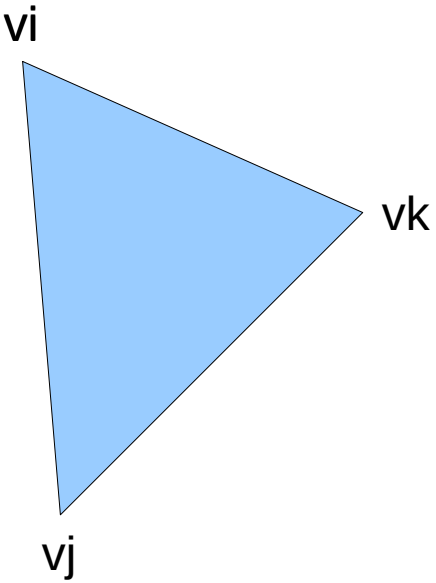




Triangle based data structure

- Essentially an (indexed) triangle soup
  - Must traverse all triangles to find neighbors

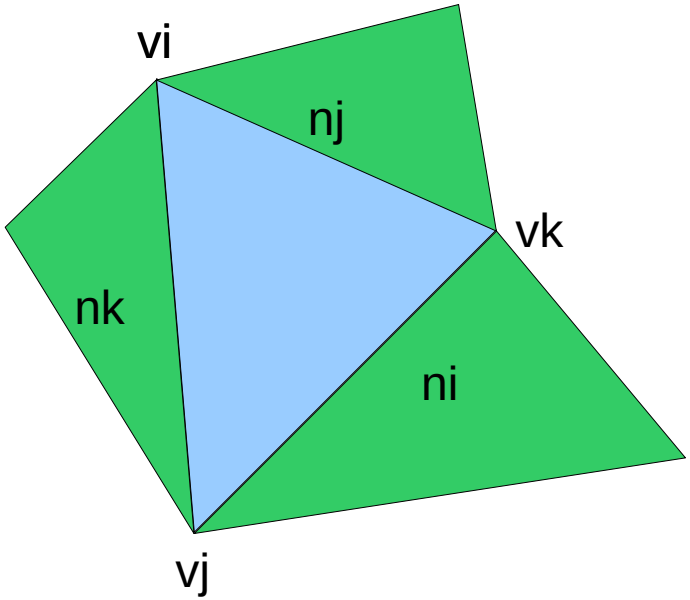
	Indices		
Triangle #	vi	vj	vk
1	1	8	2
2	2	7	4
3	2	4	6
4	7	3	2



Triangles with neighbors

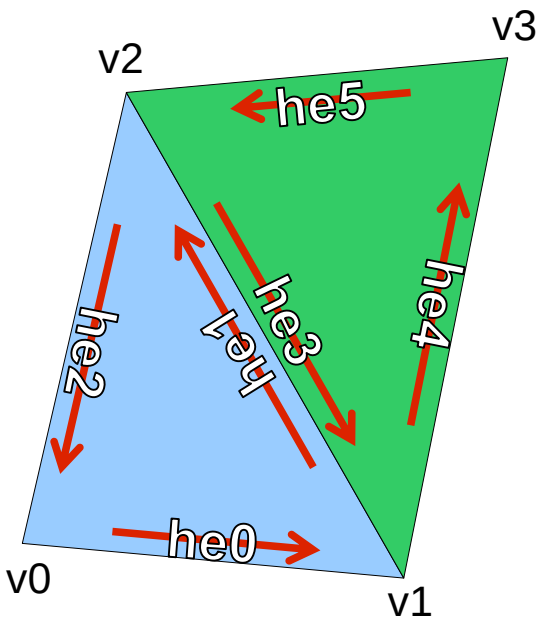
- Adds neighbor information
  - Triangle 1 is neighbor to triangles 2 and 6
  - Easy to find neighborhood.

	Indices			Neighbors		
Triangle #	vi	vj	vk	ni	nj	nk
1	1	8	2	9	-	6
2	2	7	4	4	3	-
3	2	4	6	-	-	2
4	7	3	2	5	2	6



Half Edge Data Structure

- Store triangulation as a set of “half edges”
  - Each half edge is oriented with the triangle orientation
- Very easy to traverse triangulation
- Can also include pointers to faces, etc.



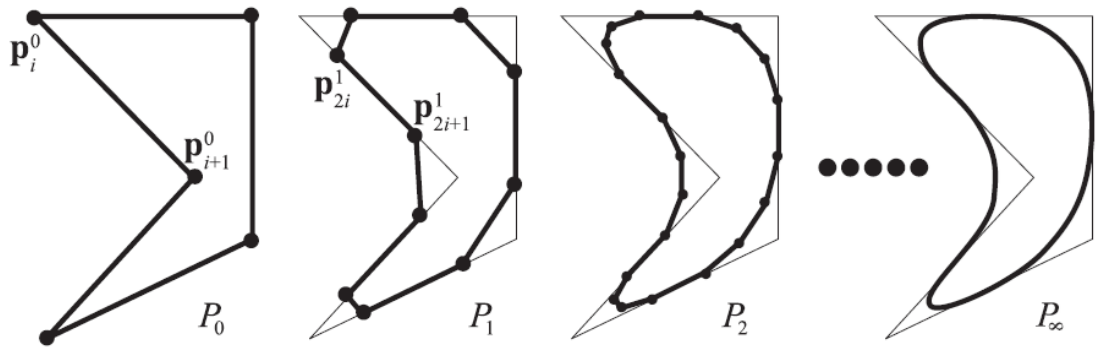
Half-Edge #	Vertex	Next HE	Twin HE
0	0	1	-
1	1	2	3
2	2	0	-
3	2	4	1

Chaikin's Scheme for Curves

- Subdivision of a curve using Chaikin's scheme
- Each edge gives rise to two new vertices
  - Original vertices are removed
  - New vertex positions are linear interpolations between two vertices at positions ¼ and ¾.
- $$p(t) = t \cdot p_0 + (1 - t) \cdot p_1$$

$$p_{2\mathrm{i}}^{k+1} = \frac{3}{4} p_i^k + \frac{1}{4} p_{i+1}^k$$

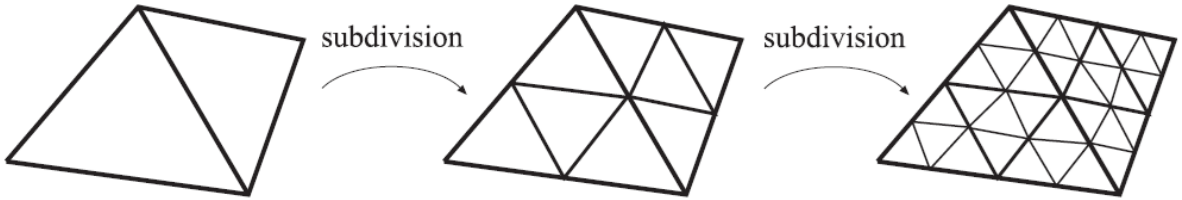
$$p_{2\mathrm{i}+1}^{k+1} = \frac{1}{4} p_i^k + \frac{3}{4} p_{i+1}^k$$



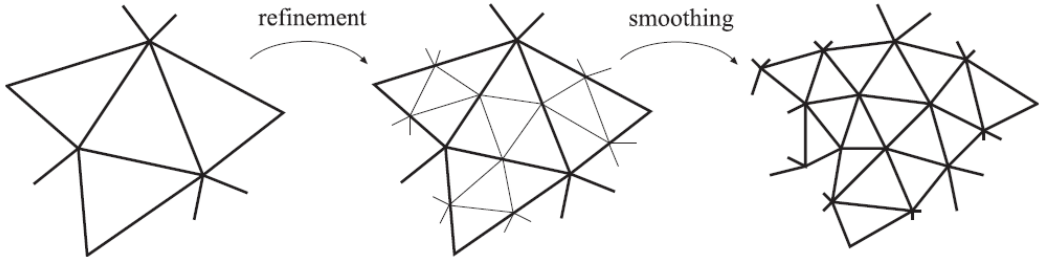
- Converges to a quadratic B-spline!

Loop Subdivision

- Creates four new triangles for each input triangle:

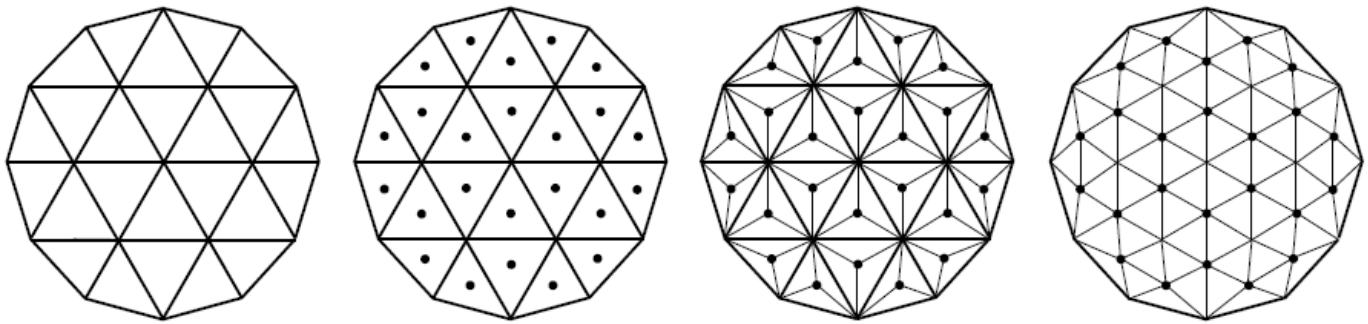


- A “two-stage” process:



  – First refine the mesh (add new vertices)
  – Then smooth (update position of original vertices)

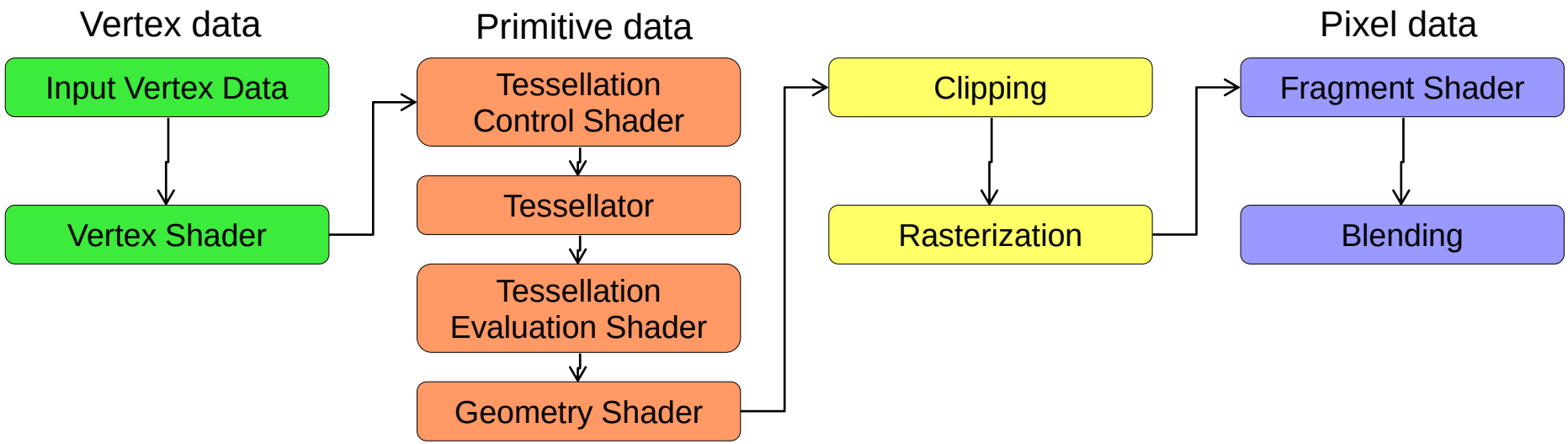
$\sqrt{3}$ Subdivision



- The first refinement stage adds new vertices at the centers of each triangle
- Triangles are created by connecting the new vertex to the existing triangle
- Then, we swap old edges
- Finally, in the smoothing step, we update old vertex positions

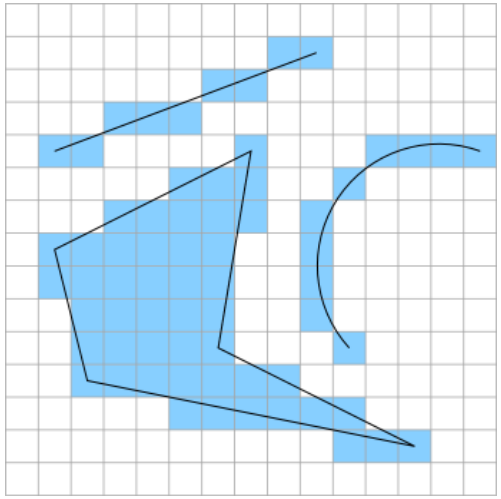
Hardware Tessellation

- Two extra shader stages:
  - Tessellation control shader: Operates on “control points”
  - Tessellation Evaluation Shader: Evaluates position of tessellated vertices



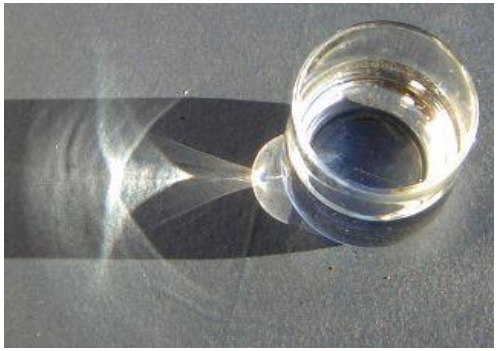
OpenGL and Rasterization

- OpenGL creates a 2D image from a 3D scene using rasterization
  - First, transform the 3D world into camera view
  - Then, rasterize: for each pixel position, get the primitive closest to the camera and shade



Rasterization Limitations

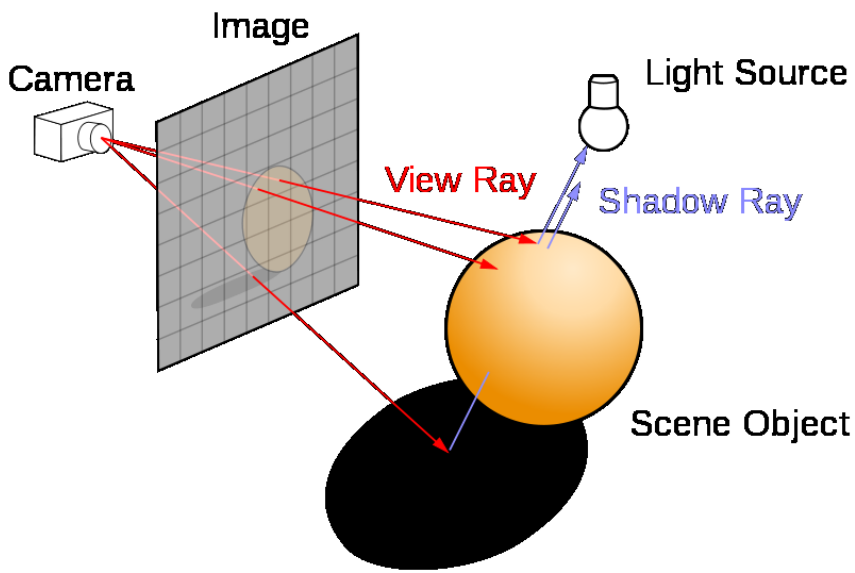
- No “secondary” effects
  - Reflections/refractins (can use (dynamic) cube maps though, see lecture 7)
  - Shadows (You all know what a pain shadows can be, right?)
  - Caustics
  - Etc.



Caustic image, Wikipedia, Heiner Otterstedt. Glass image, Wikipedia, Chmouel Boudjnah

Ray-tracing

- Ray-tracing approximates the way our eyes work
- In real life, photons hit different things, and eventually some hit our eyes
- Ray-tracing does the opposite: create a ray from the camera through each pixel, and trace it throughout the scene until it hits a light or shadow



Intersecting a sphere

- Let us use a parametric formulation for our ray:

$$q(t) = q_0 + t \cdot q_{direction}$$

- Now, we can insert into the sphere formula:

$$(q(t)_x - x_0)^2 + \left(q(t)_y - y_0\right)^2 + (q(t)_z - z_0)^2 - r = 0$$

  – Expanding out, this is a quadratic formula, where only t is unknown!
  – Solve using the quadratic formula to find intersections (roots of the polynomial)!

$$\mathrm{t} = \frac{-b \pm \sqrt{b^2 - 4\mathrm{ac}}}{2\mathrm{a}}$$

$$a = q_{direction} \cdot q_{direction}$$
$$b = 2 \cdot d \cdot (q_0 - c_0)$$
$$c = (q_0 - c_0) \cdot (q_0 - c_0) - r \cdot r$$

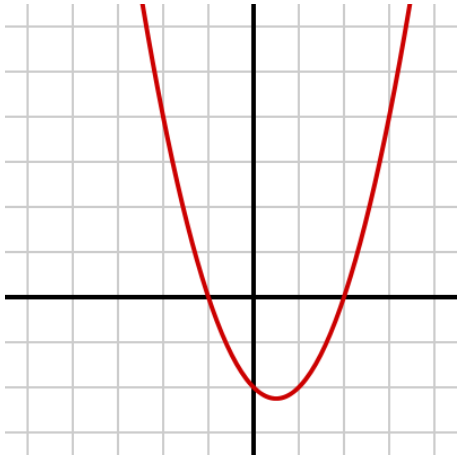
Ray-Sphere intersection

- If $b^2 - 4\mathrm{ac}$ is negative, we get an imaginary root: the ray misses the sphere

- Otherwise, we hit the sphere, and the closest intersection is the smallest positive of the two roots

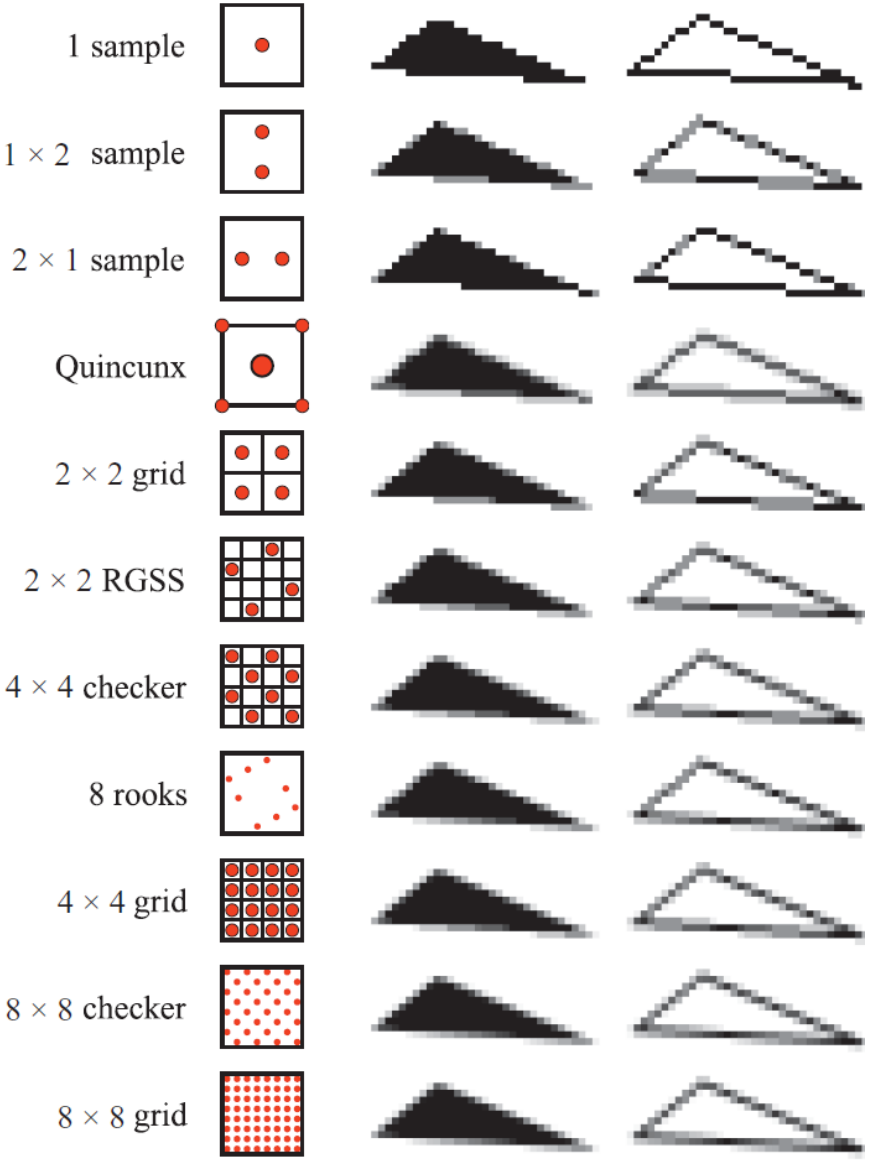
$$\mathrm{t} = \frac{-b \pm \sqrt{b^2 - 4\mathrm{ac}}}{2\mathrm{a}}$$

  - q(t) is then the intersection point
  - A negative root means that the intersection is behind the camera!
  - Both roots can be negative!



Multisampling

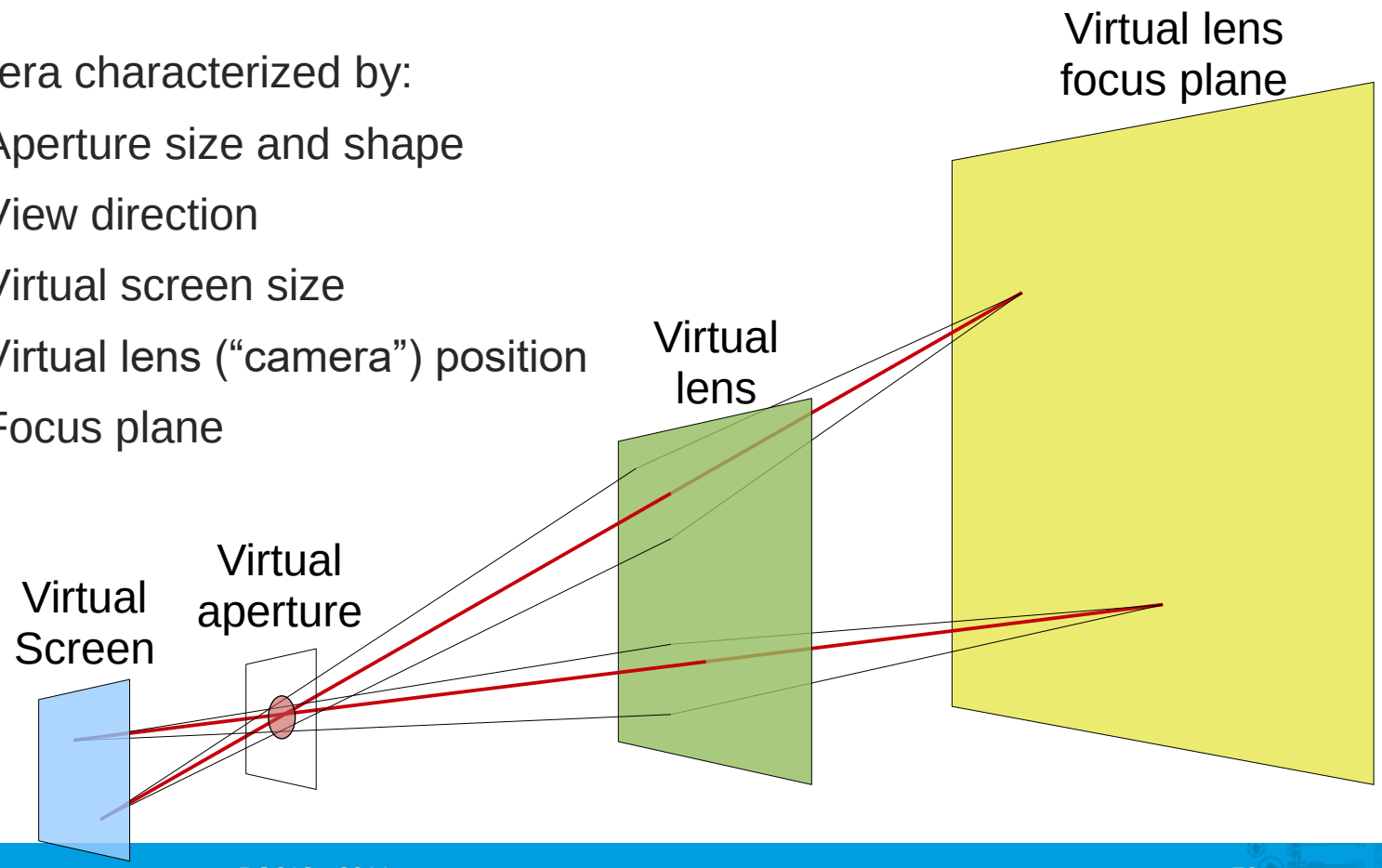
- There are many different multisampling patterns.
- We will use the 2x2 pattern
- For each pixel, instead of shooting ray at (x, y), we shoot four rays, and average their color:
  r1(x-0.25, y-0.25)
  r2(x-0.25, y+0.25)
  r3(x+0.25, y+0.25)
  r4(x+0.25, y-0.25)
  color = 0.25*(r1+r2+r3+r4)



Virtual lens camera model

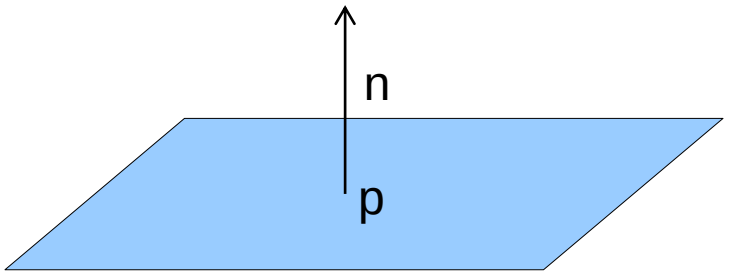
- Red lines correspond to pinhole camera, black lines give depth of field!

- Camera characterized by:
  - Aperture size and shape
  - View direction
  - Virtual screen size
  - Virtual lens (“camera”) position
  - Focus plane



Ray-Triangle Intersection

- We can also ray-trace triangle meshes
- The basic building block is the ray-triangle intersection test
- First, define the plane going through the triangle
  - Remember that a plane can be represented by a point on the plane and the plane normal

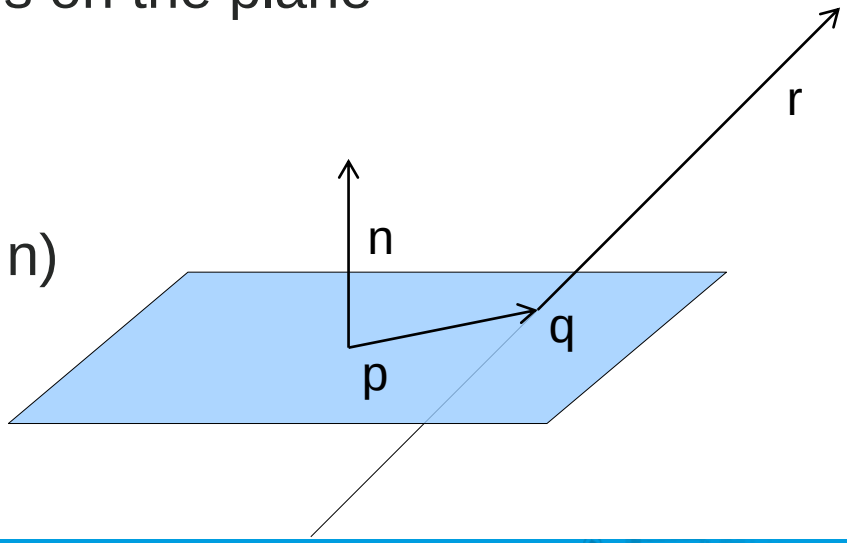


See also http://en.wikipedia.org/wiki/Line-plane_intersection

Ray-Triangle Intersection

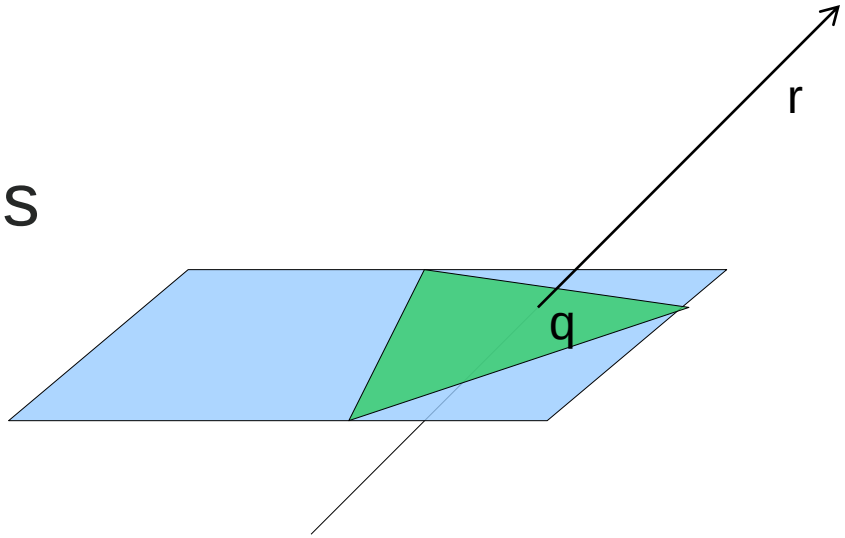
- Our ray can hit this plane
  - Zero times: runs parallel to the plane
  - Infinitely many times: runs along the plane
  - One time

- We know that (q-p) dot n = 0 if q is on the plane
  - Insert q = r.orig + t*r.dir
  - Solve for t!
  - t = (p – r.orig) dot n / (r.dir dot n)



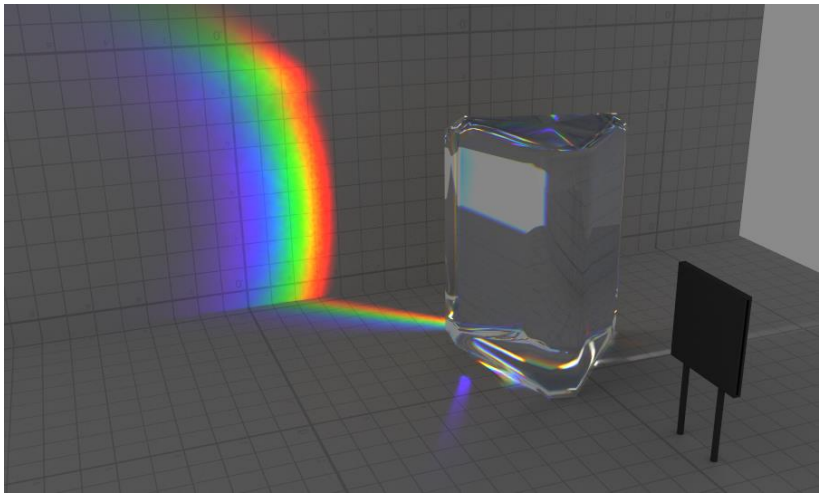
Ray-Triangle Intersection

- We now know the point in the plane of the triangle.
- We need to compute if it is within the triangle boundaries
  - Three *half space* tests
  - Barycentric coordinates



Global illumination techniques

- Photon mapping,
- Radiosity,
- Ambient occlusion,
- Etc.

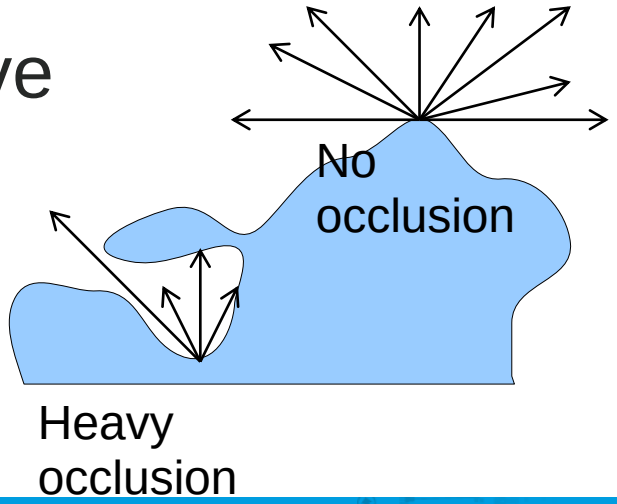


Photon Mapping

- Photon mapping is “ray-tracing for lights”
- The idea is very simple
  - Two passes
  - Pass 1: Ray-trace from the light, and note every intersection a light ray hits in the photon map
  - Pass 2: Ray-trace from camera. For shading, query the stored locations from the photon map: sum up the light contribution for the photons

Ambient Occlusion

- Gives good perception of shadows
- Simple idea: For each point to be shaded: how much of the “sky” can you see?
  - The less “sky” you can see, the darker the shadow you are in
  - Occluders close to you that give strong “shadow” contribution
  - Occluders far away give less “shadow” contribution

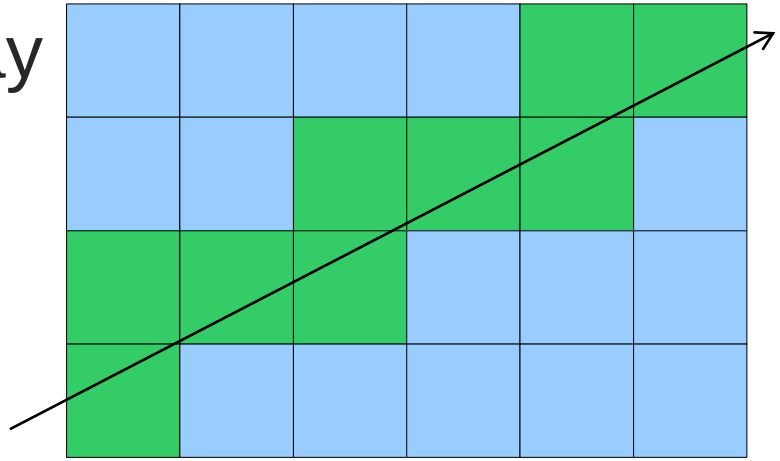


Screen Space AO

- Ambient Occlusion can also be done in screen space (OpenGL post process on a 2D texture)
- Render the scene to a texture and store color, depth, and normal data
- In a separate pass:
  - For each pixel, shoot random rays along the normal hemisphere
  - If you hit neighboring pixels that have less depth (in normal direction), they contribute to occlusion

Volume Rendering: Ray-Casting

- Simple in principle:
  - Fire a ray per pixel as in ray-tracing
  - This ray will hit the voxels along its path
  - Gather the color for each voxel we visit, and blend using the length of our ray within the voxel
  - Stop tracing when all transparency is “used”



Cut Planes

- Using many cut planes, this gives good results

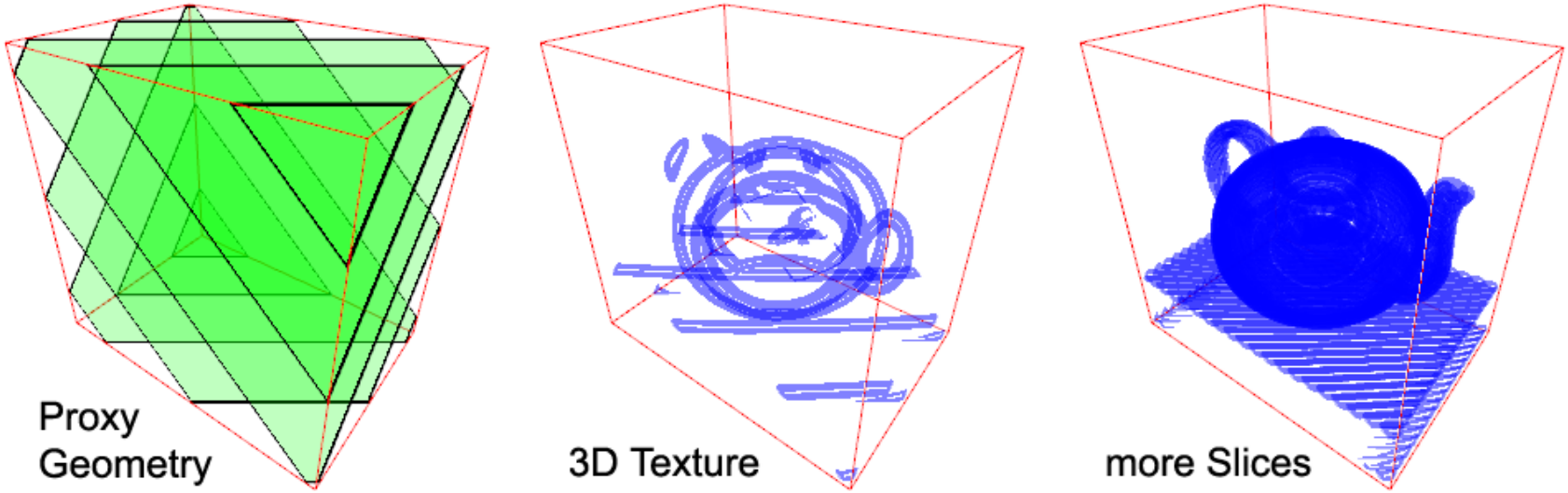


Image from Direct Volume Rendering, Matthias Bolte, Olaf Lüke, Bastian Nordmeyer

Optimization

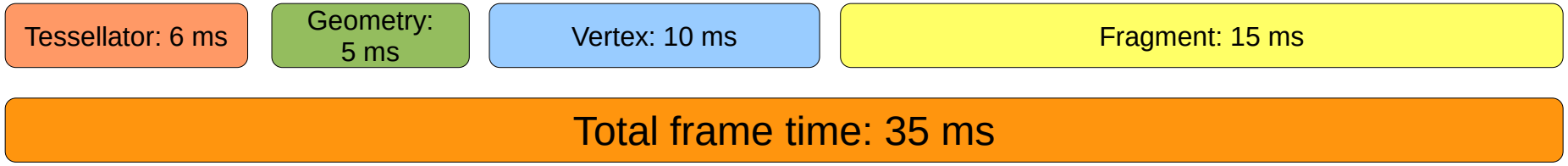
- “Premature Optimization
  is the root of all evil” – Knuth

- Make it work first,
  then optimize!

- Optimization is mainly:
  - Locate bottleneck
  - Optimize
  - Repeat

Identifying Shader Bottlenecks

- Let us assume that one of the shader stages is the bottleneck
- The frame time is then limited by the slowest shader stage
- Optimizing the slowest stage will increase performance the most!



Lab

- Implement a Mandelbrot renderer from the sample code!
  - The sample code displays the x/y screen coordinate as a color
  - Implement the Mandelbrot set
  - Use the x-coordinate as the real part of the complex coordinate, and the y-coordinate for the imaginary
  - See also the wikipedia article for help
- Discussion: How can we optimize?

Lab – Konkurranse:D!

- Tips: Sitt to og to og samarbeid
- Tre premier!
  - Premie 1: Nidar Favoritter!
    Første til å implementere mandelbrot med RGB shading
  - Premie 2: Firkløver (pensjonistsjokoladen)!
    Første til å implementere smooth HSV-interpolasjon
  - Premie 3: Stratos!
    Andre til å implementere smooth HSV-interpolasjon
- PS: Hver person kan maks vinne én premie