A Trip Down The (2003) Rasterization Pipeline

Aaron Lefohn - Intel / University of Washington
Mike Houston – AMD / Stanford

Acknowledgements

In addition to a little content by Aaron Lefohn and Mike Houston, this slide deck is based on slides from

- Tomas Akenine-Möller (Lund University / Intel)
- Eric Demers (AMD)
- Kurt Akeley (Microsoft/Refocus Imaging) - CS248 Autumn Quarter 2007

This talk

- Overview of the real-time rendering pipeline available in ~2003 corresponding to graphics APIs:
  - DirectX 9
  - OpenGL 2.x

- To clarify
  - There are many rendering pipelines in existence
    - REYES
    - Ray tracing
    - DirectX11
    - …
  - Today's lecture is about the ~2003 GPU hardware rendering pipeline

If you need a deeper refresher

- See Kurt Akeley's CS248 from Stanford
  - http://www-graphics.stanford.edu/courses/cs248-07/schedule.php
  - This material should serve as a solid refresher
- For an excellent "quick" review of programmable shading in OpenCL, see Andrew Adams' lecture at the above link
- GLSL tutorial
  - http://www.lighthouse3d.com/opengl/glsl/
- Direct3D 9 tutorials
  - http://www.directxtutorial.com/
  - http://msdn.microsoft.com/en-us/library/bb944006(v=vs.85).aspx
- More references at the end of this deck

The General Rasterization Pipeline

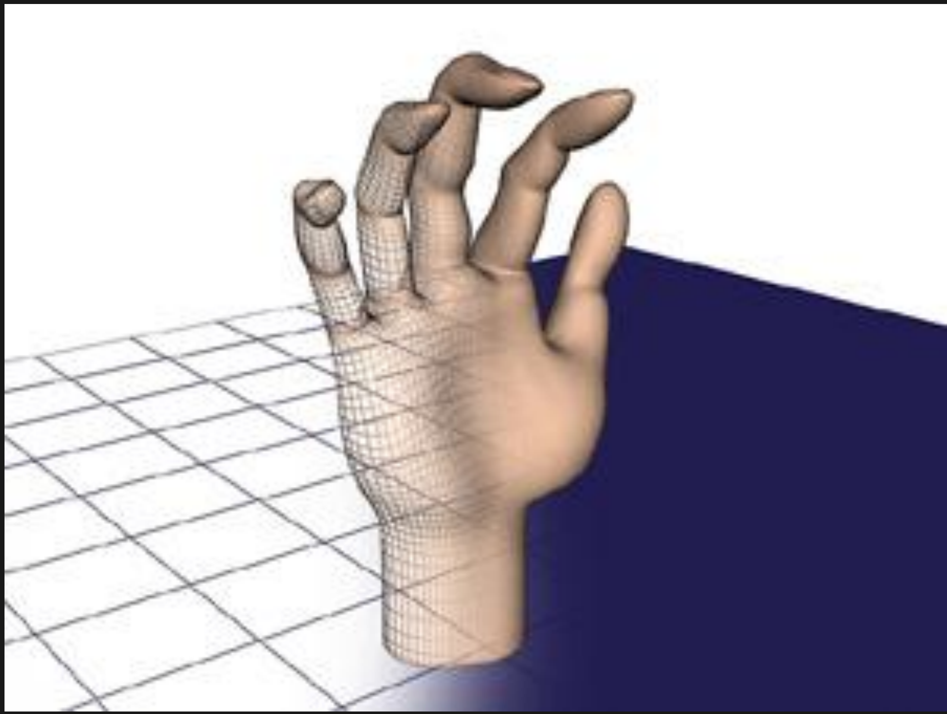
Rendering Problem Statement

- Rendering is the process of creating an image from a computer representation of a 3D environment using algorithms that simulate cameras, lights, reflections

- Rendering is an insatiable consumer of compute resources and memory bandwidth
  - Long history of special hardware created to accelerate rendering
  - Rendering is great at consuming *all* available compute resources

Objects and 3D space

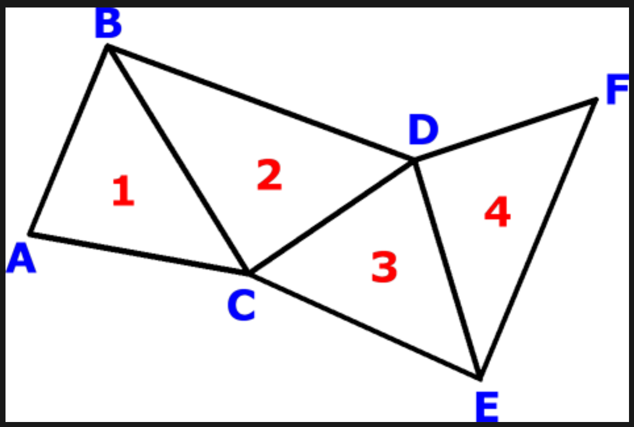
- A virtual space is created within a computer
  - Space has all 3 geometrical dimensions and, optionally, time
- Objects within the space exist and are composed of geometric primitives and their parameters (colors, surface properties)
  - Primitives are simply points, lines, triangles and perhaps higher order surfaces

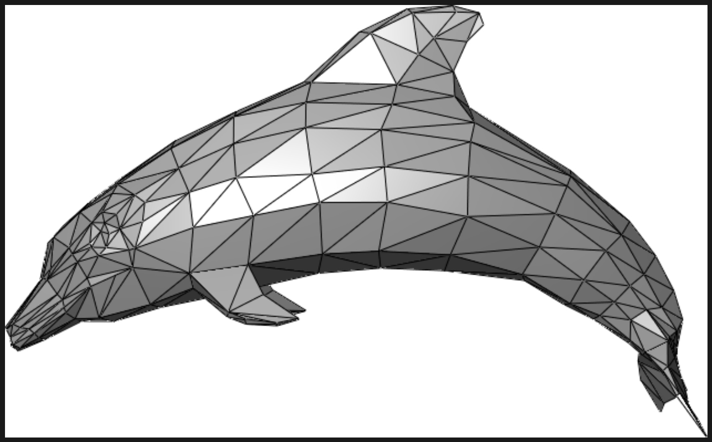


"How 3d graphics works" – How stuff works – Curt Franklin

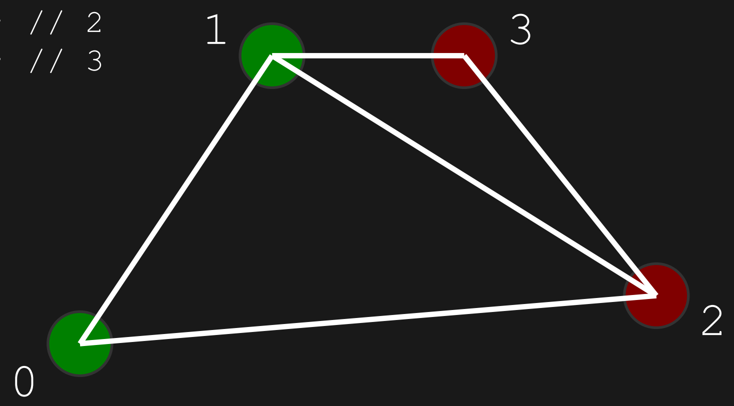
The Primitive

- A collection of vertices to create points, lines, triangles, strips of triangles, and meshes
- Attributes within the primitives come from the vertex attributes
- Example uses OpenGL syntax





```
glBegin(GL_TRIANGLE_STRIP);
glColor(green);
glVertex2i(…); // 0
glVertex2i(…); // 1
glColor(red);
glVertex2i(…); // 2
glVertex2i(…); // 3
glEnd();
```



The Primitive parameters

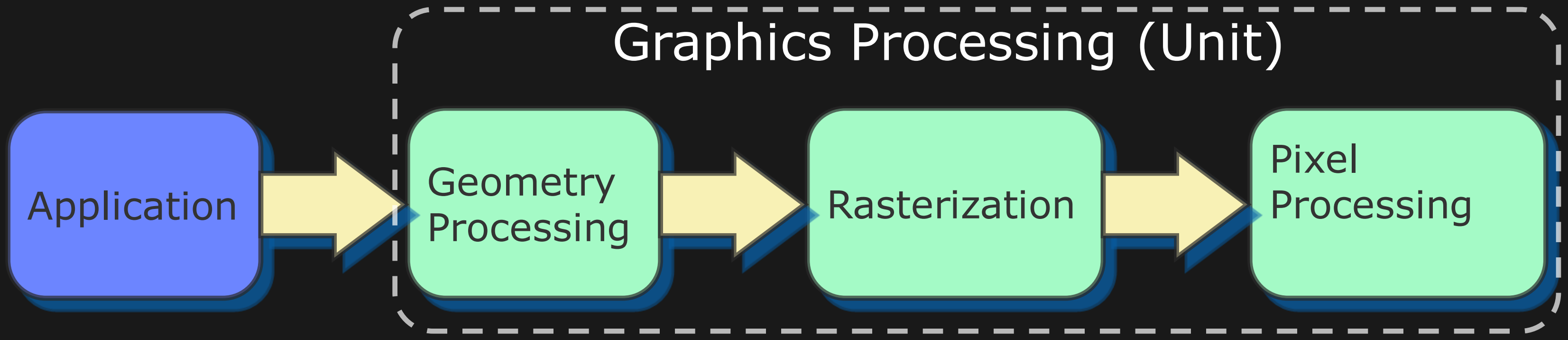
- Beyond geometry, primitives also have other parameters beyond (XYZW) and also offer color and texture coordinates.
- An example of colors per vertex and simple shading:

```
glBegin(GL_POLYGON);
  glColor(RED);
  glVertex3i(0,0,0);
  glVertex3i(1,0,0);
  glVertex3i(0,1,0);
glEnd()
```

```
glBegin(GL_POLYGON);
  glColor(RED);
  glVertex3i(0,0,0);
  glColor(BLUE);
  glVertex3i(1,0,0);
  glColor(BLUE);
  glVertex3i(0,1,0);
glEnd()
```

Akeley, Hanrahan [3]

General Rasterization Pipeline



- Geometry processing:
  - Transforms geometry, generates more geometry, computes per-vertex attributes
- Rasterization:
  - Sets up a primitive (e.g., triangle), and finds all samples inside the primitive
- Pixel processing
  - Interpolates vertex attributes, and computes pixel color

Slide by Tomas Akenine-Möller

Rasterization vs Ray Tracing

- Given that many of you have written a ray tracer but not a rasterizer…

- A rasterization rendering pipeline can be thought of as a special-purpose ray tracer that is highly optimized to only trace rays that:
  - Share a common origin
  - Terminate at the first intersection

Rasterization vs Ray Tracing

- Rasterizer

```
Foreach triangle, t {
   Foreach pixel that intersects t {
      … }}
```

Ray tracer

```
Foreach pixel, p {
  Foreach triangle that intersects p {
     … }}
```

- Parallelization and optimizations are quite different

- The two are often combined in the same rendering

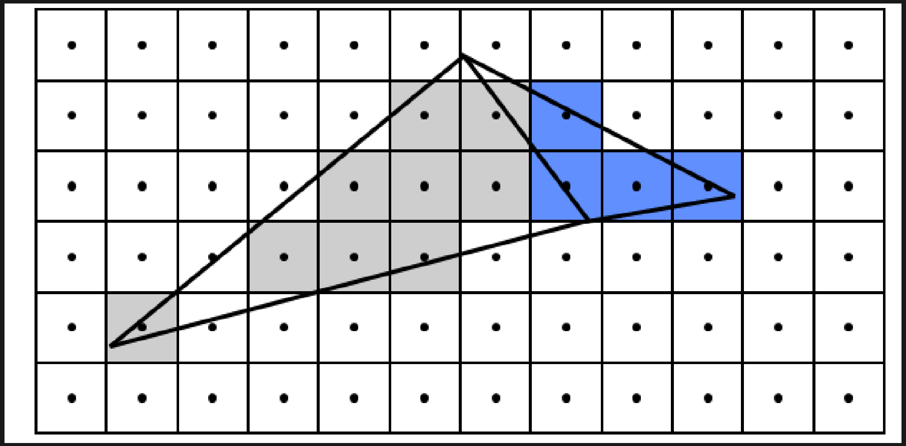
Rasterization – some definitions

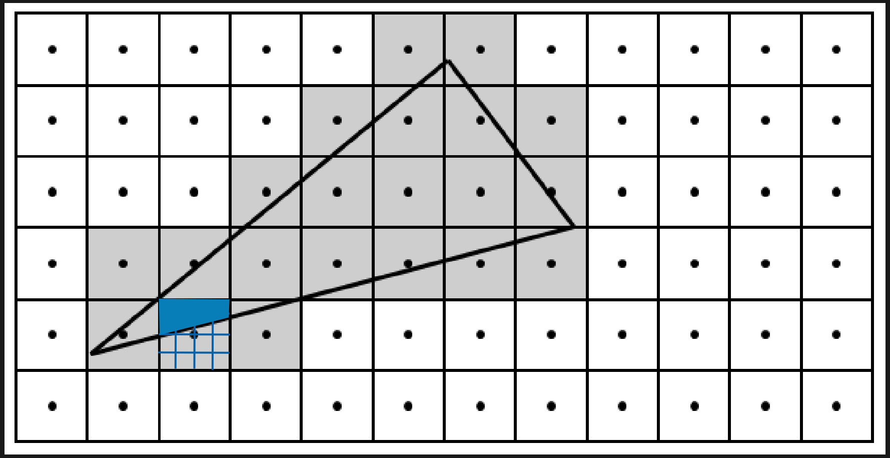
- A “Pixel”
  - Short for “picture” “element”
  - Smallest visible unique element on a display
    - But internally in the pipe, some elements could be smaller

- Rasterization
  - Rasterization is the act of generating visible pixels from primitives, through scan conversion, texturing, shading, color blending, etc.
    - Basically, to identify which pixels to “light up” and what they look like
  - A “raster”, or Latin’s *rastrum*, is a rake. This loosely translates to a device that draws parallel lines, or a grid of squares – in our case, pixels.

Rasterization & Scan Conversion

- "Scan conversion" is the act of finding which screen pixels belong to a given primitive
- Rules of rasterization vary, with multiple techniques and rules dependent on API, algorithm
- The part of the primitive that is within (totally or partially) a pixel is called a "fragment". It's defined with a pixel and a coverage
- Pixels are generally subdivided into a grid, and sub-grid elements are identified to be part of a primitive, and the union is a fragment.

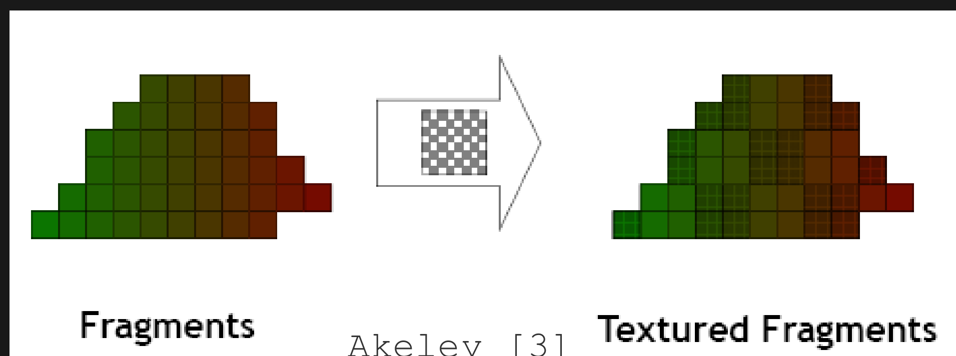




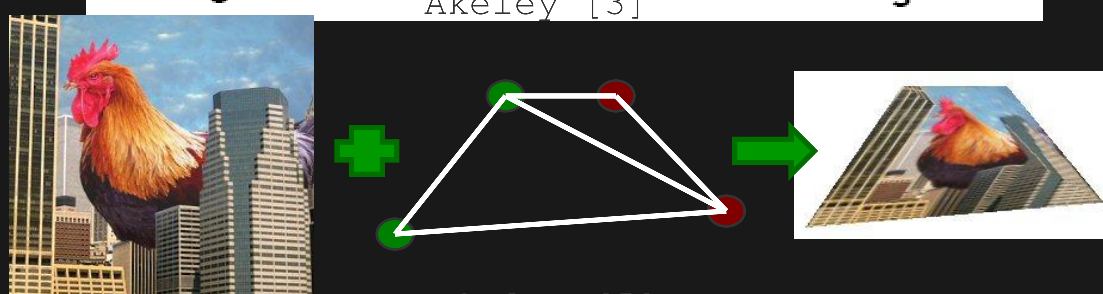
Akeley [4]

Texturing

- Texturing is the act of taking an image and painting it onto primitives or object surfaces.
- Texture “maps” are the sources images, and each picture element in the texture map is called a “texel” (similar to pixel but on the source)
  - A map is effectively a digital image with (nxm) picture elements, called “texture elements”
- The act of mapping must do a proper projection from texture space to pixel space
  - Vertices are specified in both geometry and texture space
- Texture operations may occur in fragment/pixel processing stage, but also in vertex processing (texture displacement of vertices)



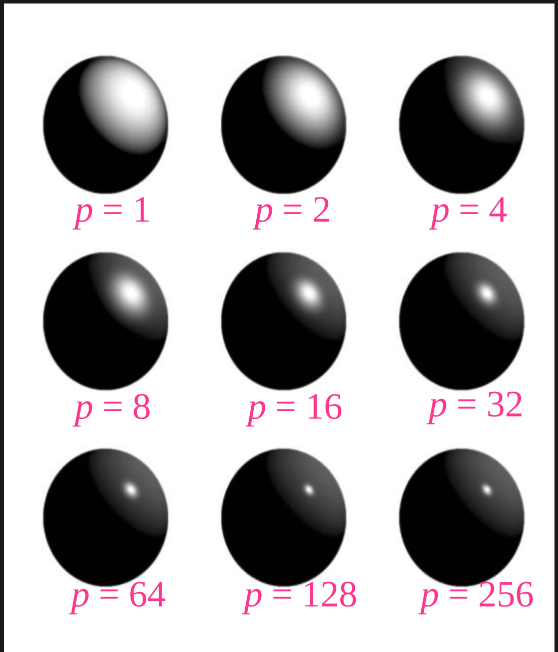
Akeley [3]



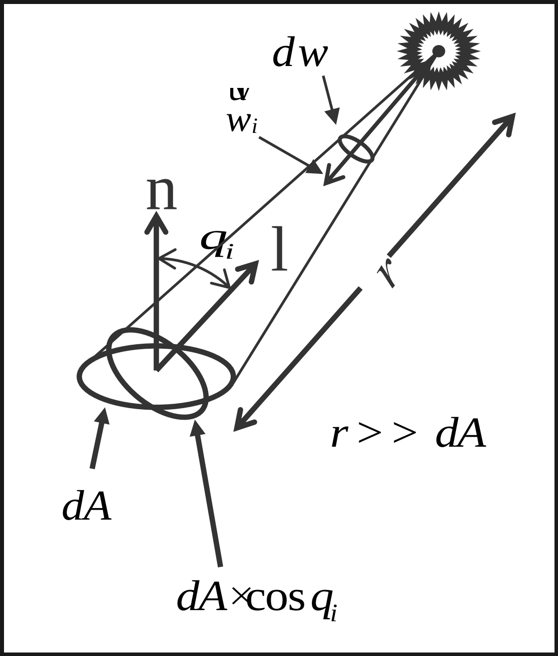
Akeley [5]

Lighting

- Realistic lighting must include the perception of light and all its effects
- Lighting may be computed per-vertex or per-pixel
  - Note: OpenGL 1.x / DirectX1-7 only permitted per-vertex lighting



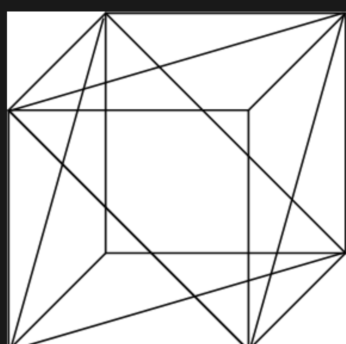
Merrell [8]



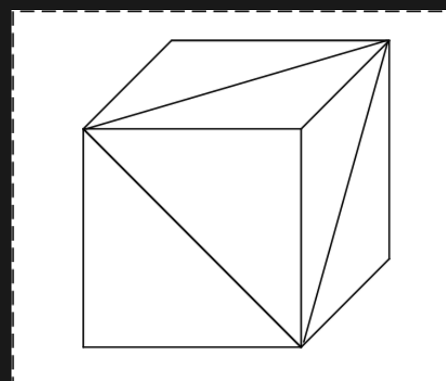
Akeley [9]

Hidden surface removal

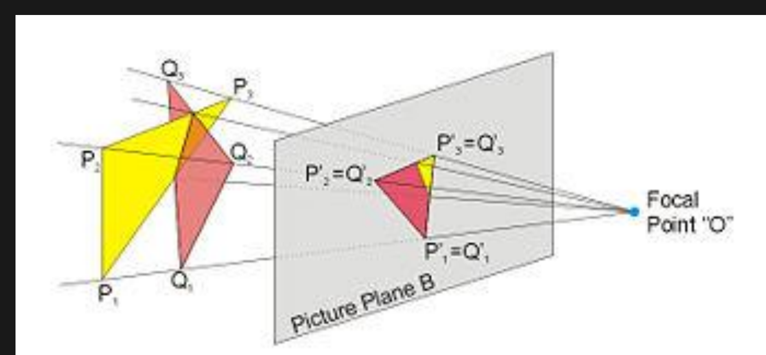
- The goal of rendering is to build a mathematical model of a world in 3D, then draw that world from the point of view of a single camera positioned arbitrarily
- Z-Buffering is a process of cutting out renderings that cannot be seen by the viewpoint of the screen using a depth process – "Z" commonly is the depth parameter to determine what shapes are in front or behind. There are other culling "tricks" used, such as back face primitive removal and others.
- Z-Buffering deals with depth on a per pixel basis, and each pixel has a depth in a "Z" buffer. This allows removal of hidden surfaces even when surfaces intersect.



Rendered virtual space



What the screen will see



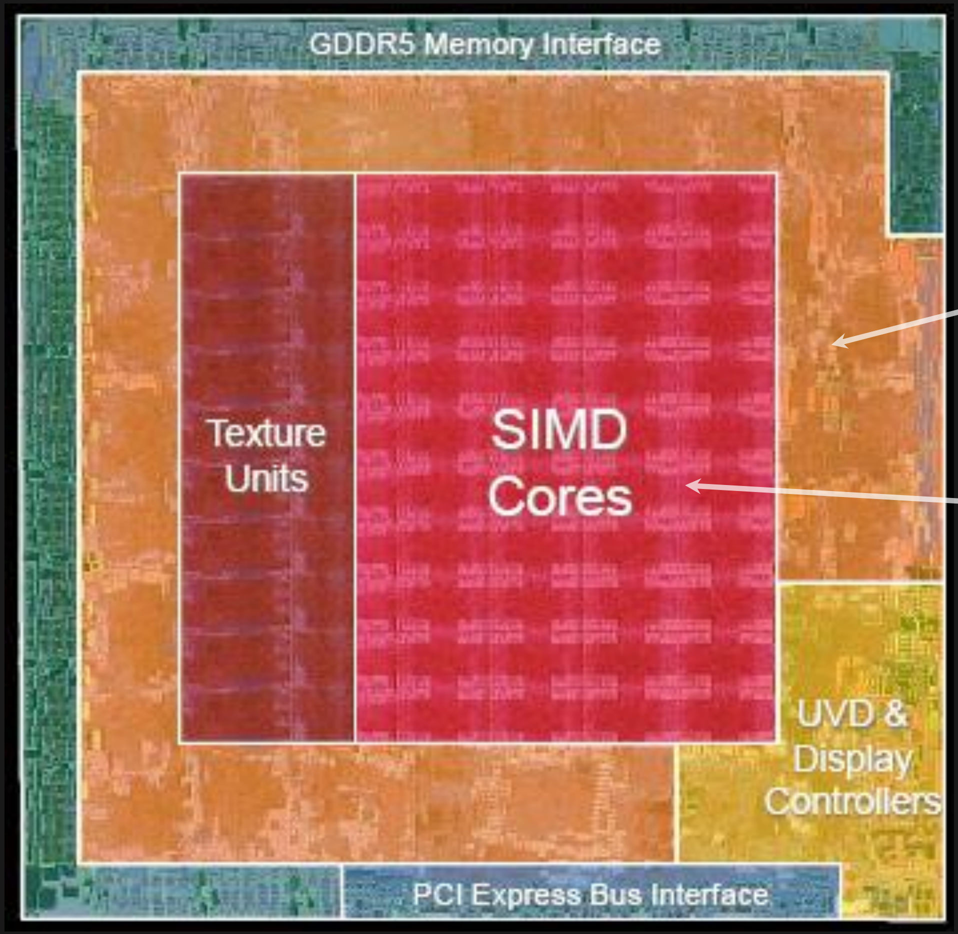
The DirectX9 / OpenGL 2.x Rasterization Pipeline

What is a GPU?

- Traditional definition
  - The Graphics Processing Unit is the hardware in a compute system that is used to generate all the contents that will be displayed on a monitor
  - This unit may come in many forms – from chipsets integrated into a default motherboard configuration to “discrete” cards which are dedicated higher-performance hardware for display driving.
  - A typical computer system will have a GPU connected to the host computer through an interface VGA/DVI/DP, and will be accompanied by memory, often called “frame buffer”.

- Integrated CPU-GPU are changing this slightly
  - GPU and CPU are now on same die and share parts of memory system
  - Division of labor between CPU and GPU for rendering is changing
  - (Intel Sandybridge and AMD Fusion)

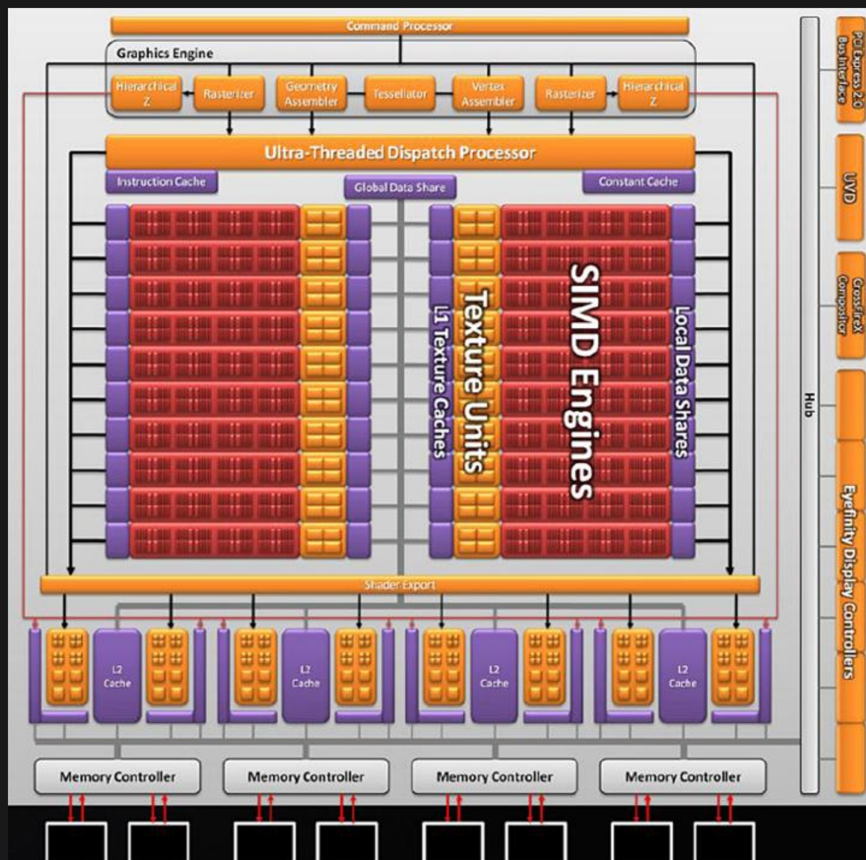
AMD GPU example HD4870, ca. 2008 (actually a DX10 GPU)



Fixed function graphics HW

Compute cores

NVIDIA GPU example (a DX10 GPU)

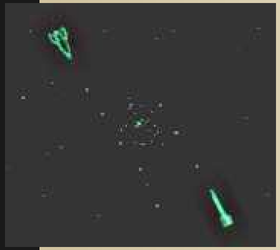


A quick history 1960’s to 2000s

Good reference: http://hem.passagen.se/des/hocg/hocg_1960.htm

1960’s

- *ISutherland, Sketchpad, 1961*
- *S Russel, Spacewars, 1961*
- IBM2250, 1965
- *Odyssey, Magnavox, 1966-8*
- U Utah, 1968
- Hidden surface removal, UU, 1969
- E&S, 1968
- *Intel, 1968*
- *AMD, 1969*



1970’s

- *Gouraud, 1971*
- Pong, Atari, 1972
- *“Westworld” PDG, 1973*
- Siggraph ACM, 1973
- Z, Texture, Catmull, 1974
- NYIT, 1974
- Apple II, Apple, 1977
- Lucas CGD, 1979

1980’s

- *RayTracing, Whitted, 1980*
- *TRON, Disney 3I, MAGI, NYIT, 1980*
- REYES, Lucas, 1981
- *SGI, J. Clark, 1982*
- Wavefront, 1984
- *Radiosity, Cornell, 1984*
- 1st MAC wGUI, Apple, 1984
- Pixar, Lucas, 1986
- Renderman, Pixar, 1988
- *ATI, 1985*

1990’s

- Win 3.0 wGUI, MS, 1990
- *OpenGL 1.0, SGI, 1991*
- Toy Story, Pixar, 1992
- *Reality Engine, SGI, 1993*
- Playstation, Sony, 1995
- *Nintendo 64, N, 1996*
- Quake, ID, 1996
- Voodoo3D, 3Dfx, 1997
- TNT2/ GeF256 NV, 1999
- *ArtX, 1997*

A quick history since 2000

2000-02

- ArtX1, 1st integrated TL, 2000
- *ArtX acquired ATI, 2000*
- NV acquires 3dfx, 2000
- VIA acquires S3, 2000
- *Gamecube N, 2001*
- *Xbox MS, 2001*
- DX9 MS, 2002
- *R300 ATI, 2002*
- LoftR/2Towers, WETA, 2002

2003-5

- R4xx ATI, 2004
- R5xx ATI, 2005
- NV4x NV, 2004
- DOOM3, ID, 2004
- *Half-Life2, Valve, 2004*
- *Xbox 360, MS, 2005*
- X3/600, 1st PCIe ATI, 2004
- SLI/Xfire ATI/NV, 2005
- Alias sold SGI, 2004
- *SGI bankrupcy protection, 2005*

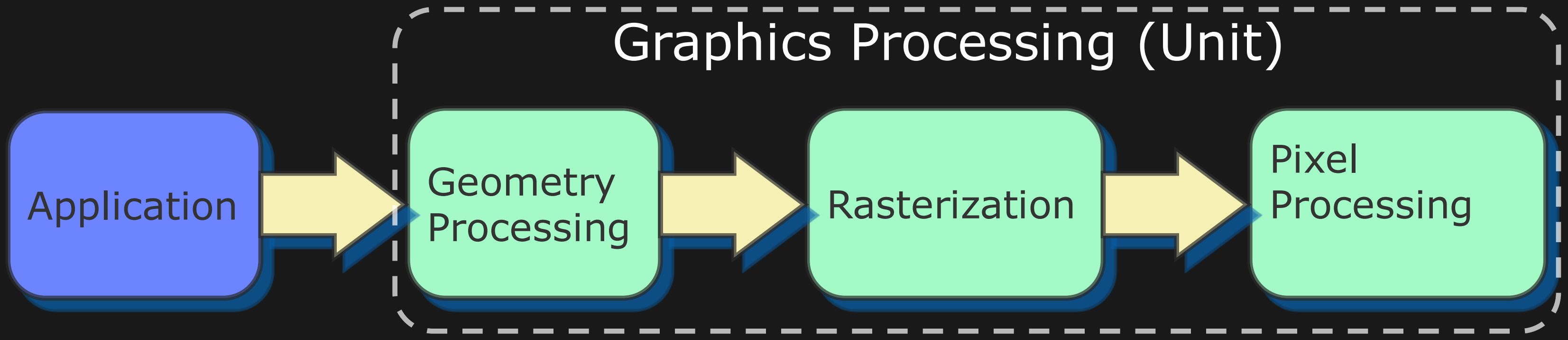
2006-7

- R6xx, ATI, 2006
- R670/x35, AMD, 2007
- G8x, NV, 2006
- *CUDA, NV, 2007*
- Vista/DX10, MS, 2006
- *Wii, N, 2006*
- PS3, Sony, 2006
- *ATI acquired AMD, 2006*
- DX 10.1, MS, 2007

2008-9

- R7xx, AMD, 2008
- G2xx, NV, 2008
- Crysis-WH, Crytek, 2008
- DX11/Win7 MS, 2009
- *OpenGL 3.0, Khronos, 2008*
- *OpenCL 1.0, Khronos, 2008*
- Evergreen, AMD, 2009
- Larrabee announcement Intel, 2008
- *SGI, Purchased, 2009*

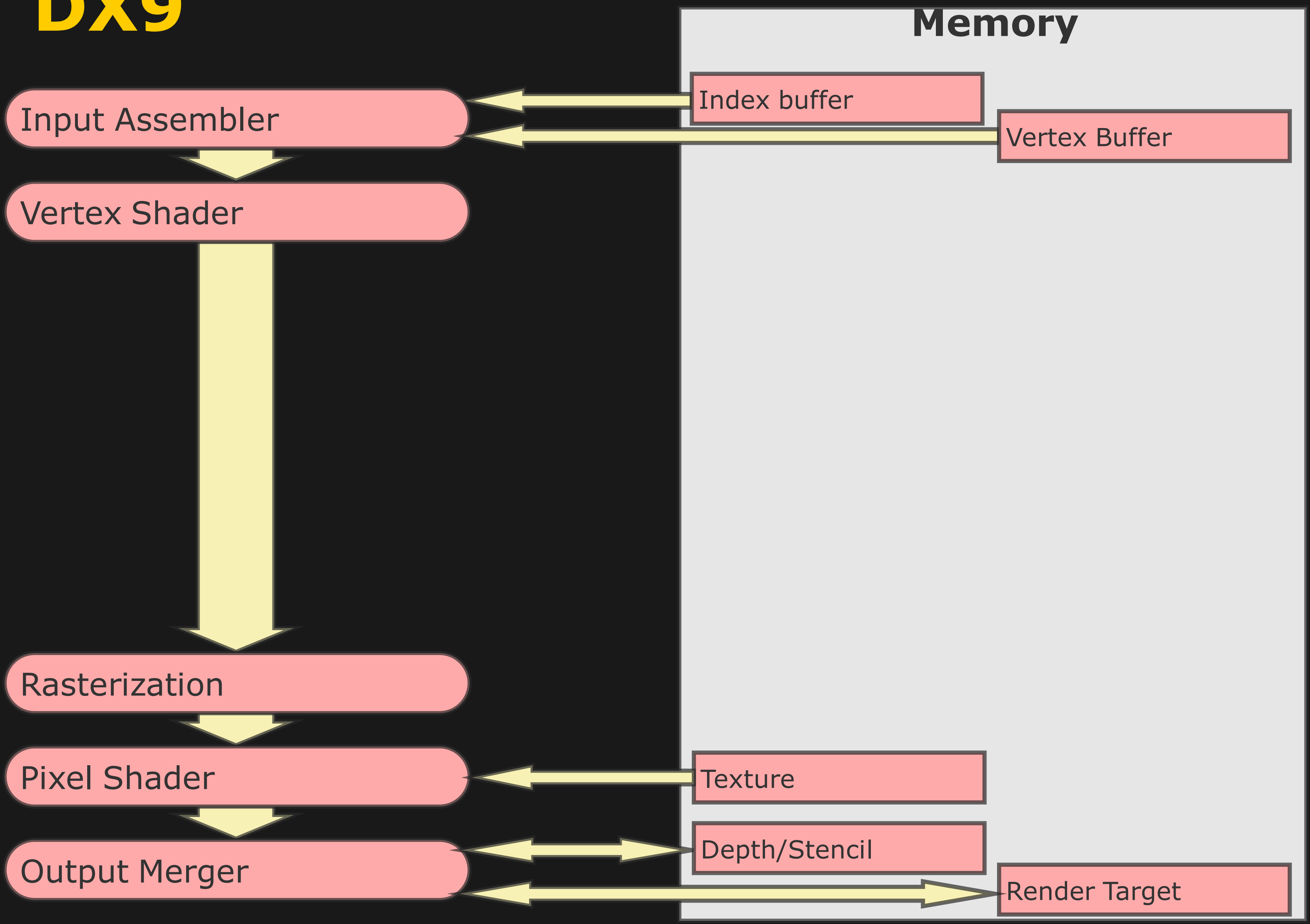
General Rasterization Pipeline



- Geometry processing:
  - Transforms geometry, generates more geometry, computes per-vertex attributes
- Rasterization:
  - Sets up a primitive (e.g., triangle), and finds all samples inside the primitive
- Pixel processing
  - Interpolates vertex attributes, and computes pixel color

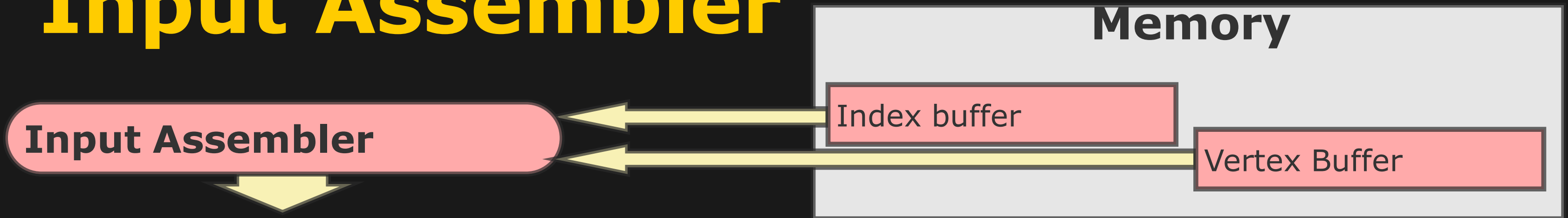
Slide by Tomas Akenine-Möller

DX9



Slide by Tomas Akenine-Möller

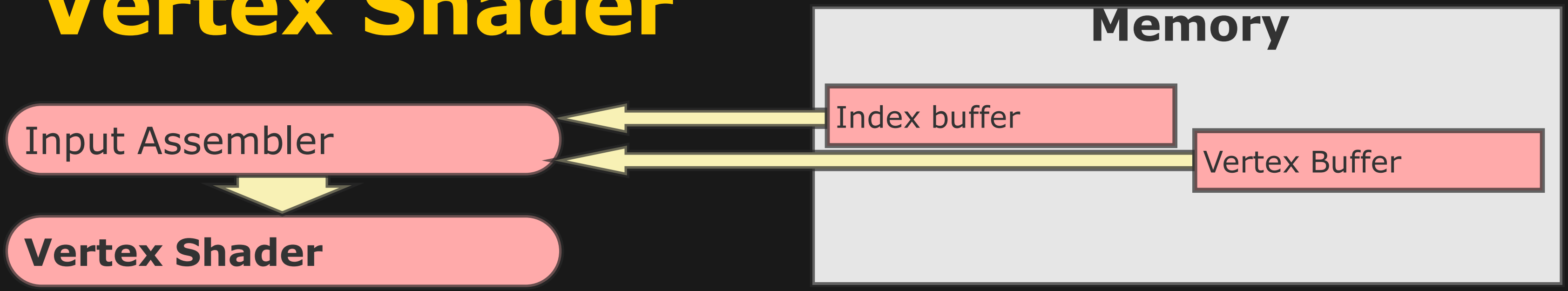
Input Assembler



- Main task is to read from memory:
  - Indices
  - Vertex attributes (xyz-coordinates, normals, texture coords, etc)
- Then form primitives (triangles, lines, points)
- Send down the pipeline

Slide by Tomas Akenine-Möller

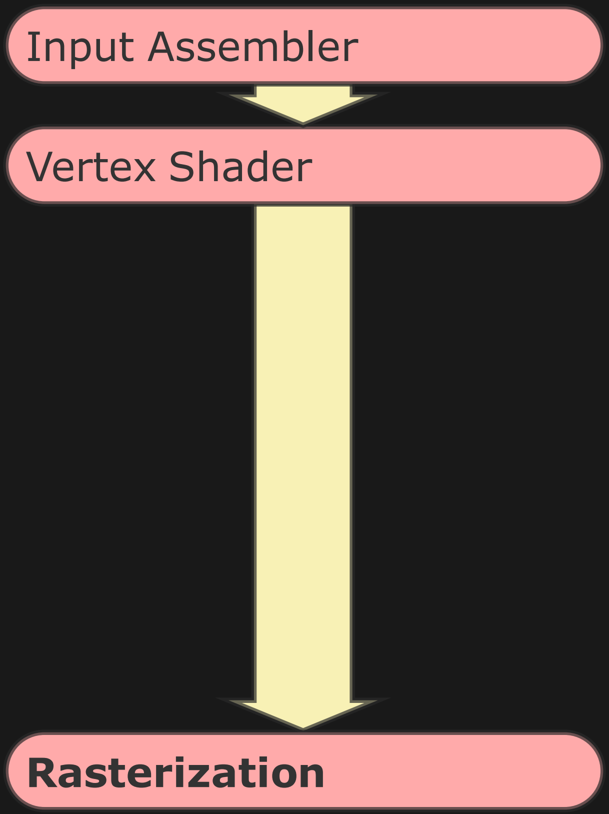
Vertex Shader



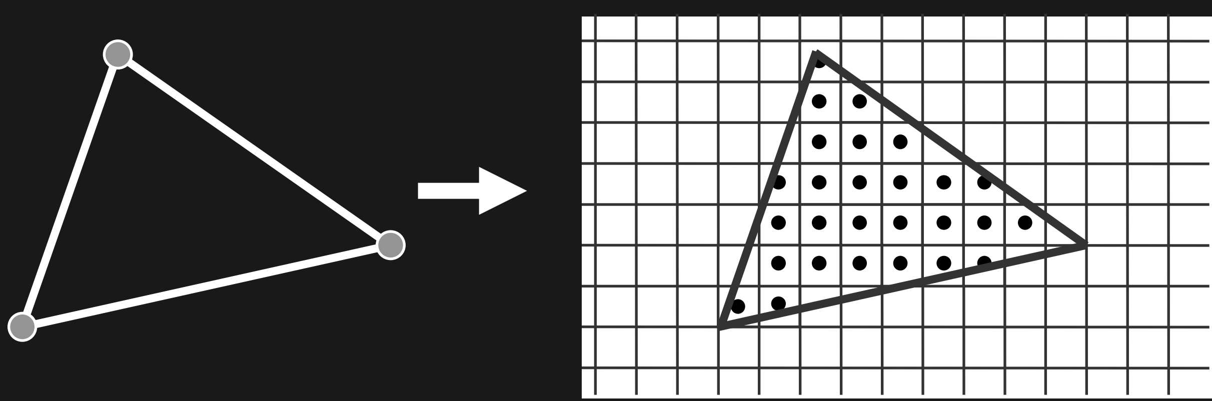
- User-supplied vertex shader program is executed once per vertex
- Examples:
  - Vertex transformations (e.g., skinning)
  - Normal/Tangent space transformations
  - Clip-space transformations
  - Texture coordinates computations (e.g., animation)
- Really up to the programmer:
  - He/she knows what interpolated attributes are needed in the pixel shader

Slide by Tomas Akenine-Möller

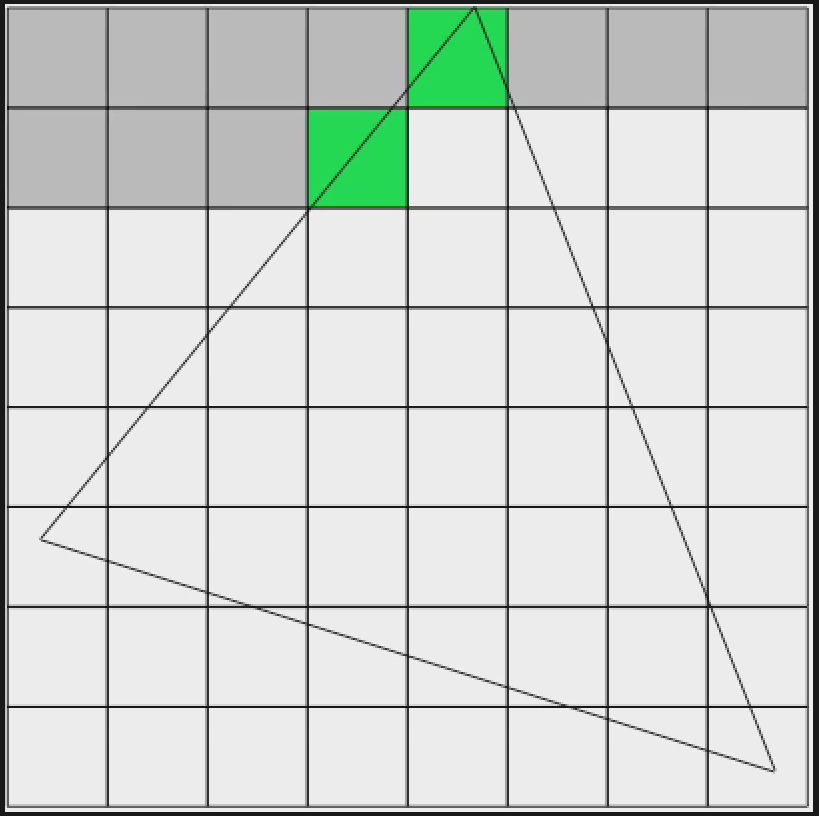
Rasterization



- Given projected vertices of a triangle:
  - find samples (one or more per pixel) that are inside the triangle

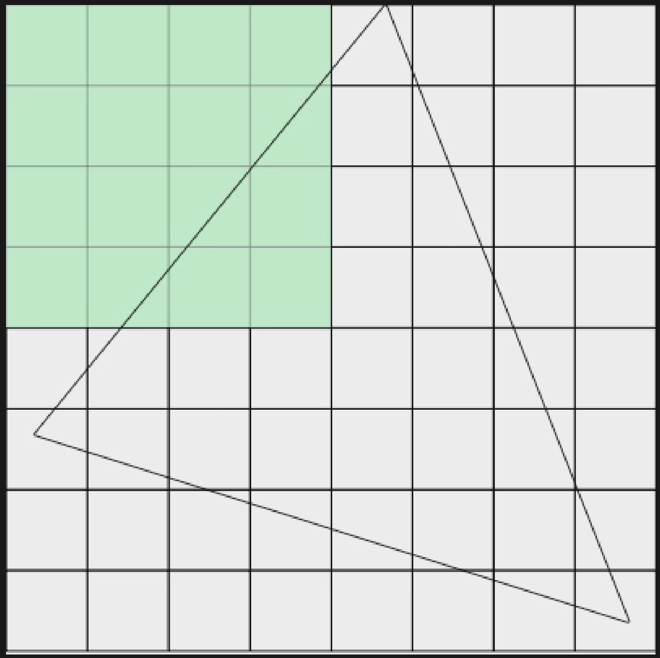


Scanline Rasterization



Slide by Tomas Akenine-Möller

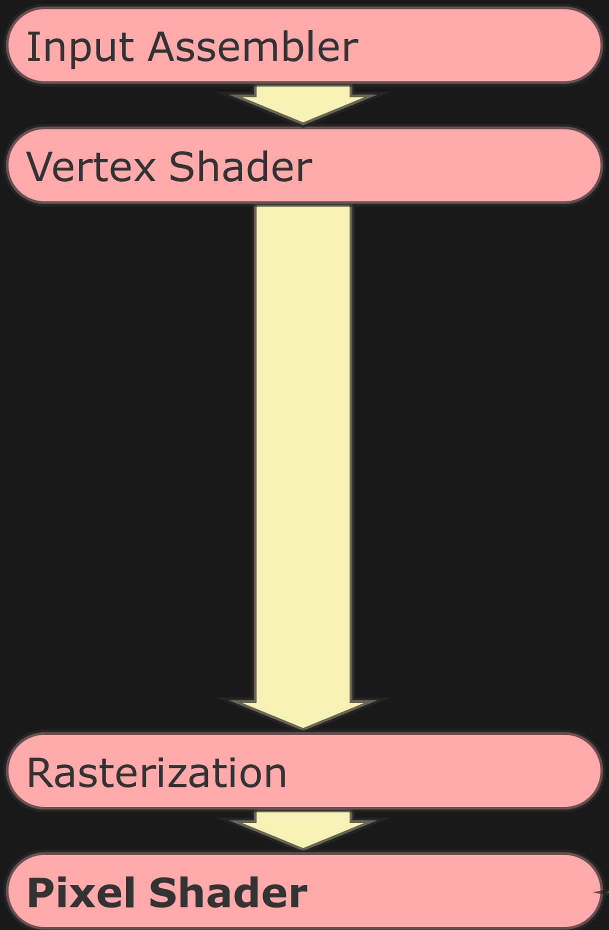
Hierarchical Rasterization



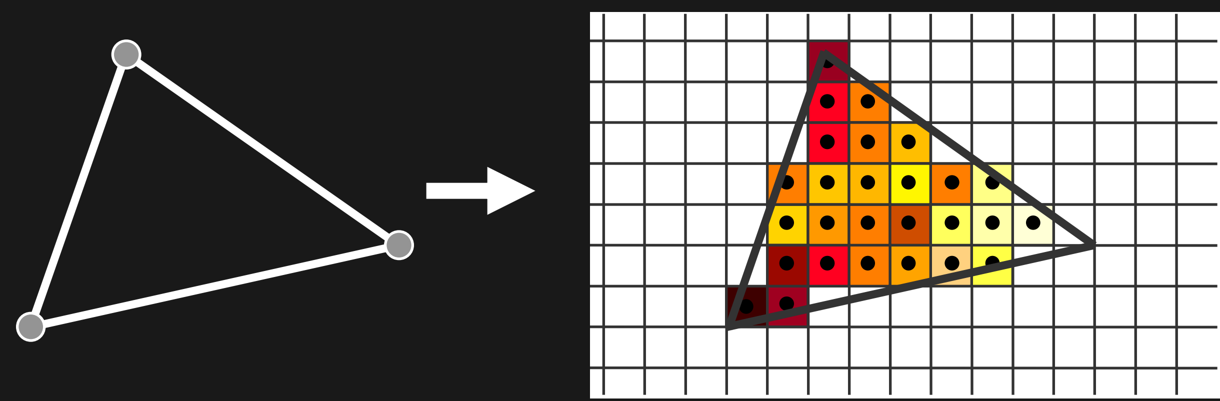
- Some variant of hierarchical rasterization is used in most real-time renderers
  - Better cache-coherence and enables z-cull, buffer compression, and exploits regularity in the problem

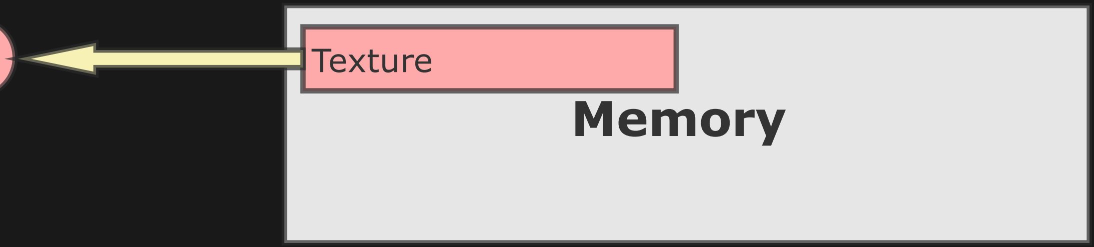
Slide by Tomas Akenine-Möller

Pixel Shader



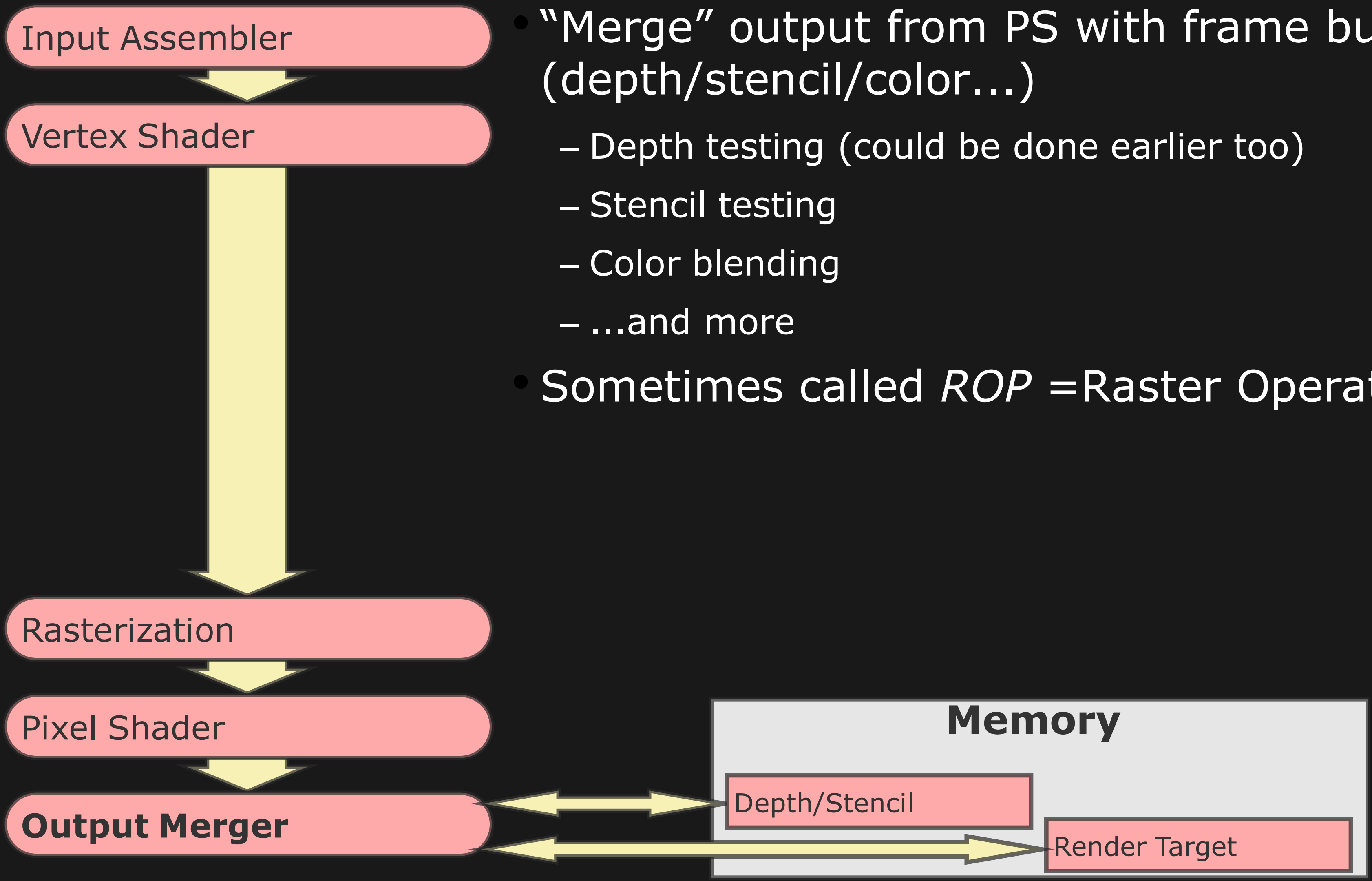
- Execute a user-supplied pixel shader program
- Task: compute pixel's color
  - - BRDF, lighting, …





Slide by Tomas Akenine-Möller

Output Merger



- "Merge" output from PS with frame buffer (depth/stencil/color...)
  - Depth testing (could be done earlier too)
  - Stencil testing
  - Color blending
  - ...and more
- Sometimes called *ROP* =Raster Operations

Slide by Tomas Akenine-Möller

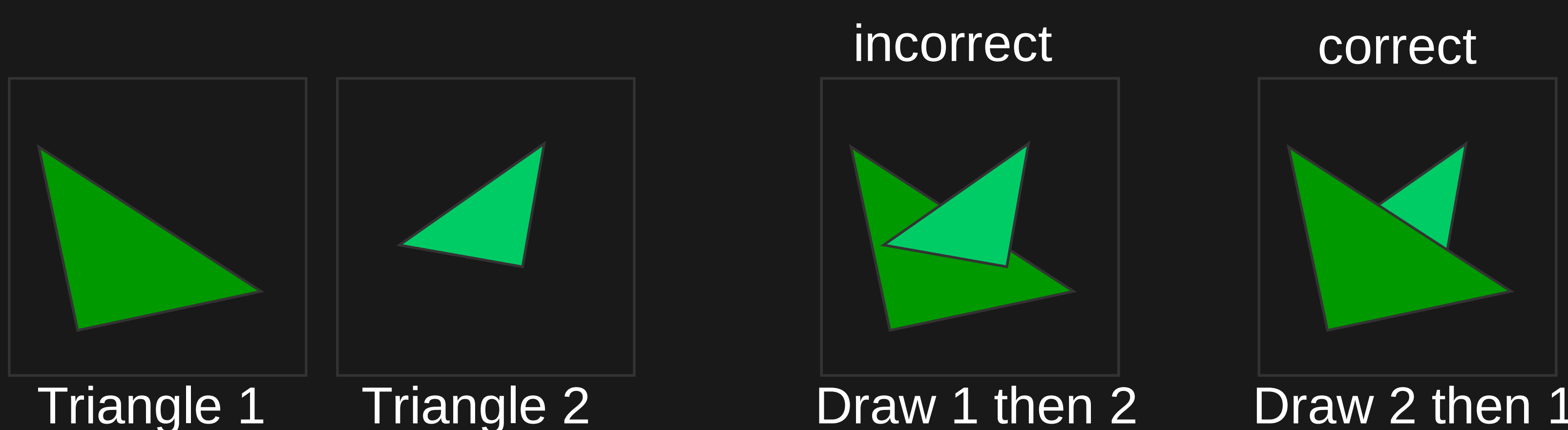
Additional Details

- Briefly about the following “standard” techniques:
  - Z-buffering (also called depth buffering)
  - Screen-Space Anti-Aliasing (e.g., MSAA, CSAA)
  - Texturing and mip-mapping
  - Z-culling

Z-buffering

Z-buffering (1)

- The graphics hardware "just" draws triangles
- A triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large triangles at different depths



Z-buffering (2)

- We need sorting per pixel
- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z-buffer z-value
  - If triangle's z is smaller, then replace Z-buffer and color buffer
  - Else do nothing
- Z-buffer characteristics
  - Render geometry in any order
  - Use fixed/bounded memory
  - Generates correct visibility result for first depth layer

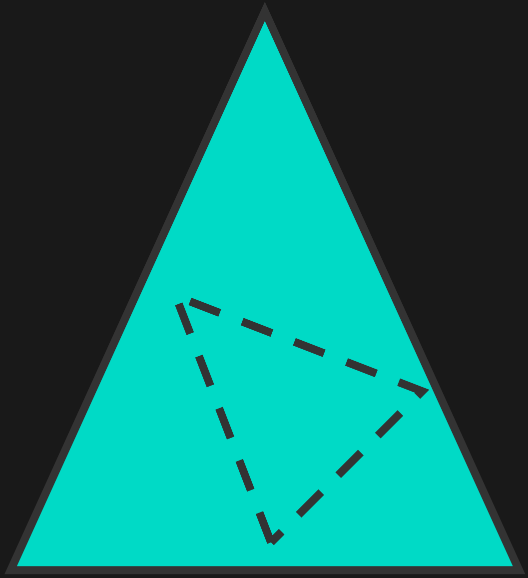
Slide by Tomas Akenine-Möller

Z-culling

Z-culling
(also called Hierarhical Depth Culling)

- Texture caching and texture compression as good ways of reducing usage of texture bandwidth
- What else can be done?

Z-Culling (aka Hierarchical Depth Culling)



- Small triangle is behind big triangle
- If this can be detected, we can:
  - reduce depth buffer accesses
  - reduce pixel shader executions
- Commonly used technique in GPUs

Screen-Space Anti-Aliasing
(including MSAA/CSAA)

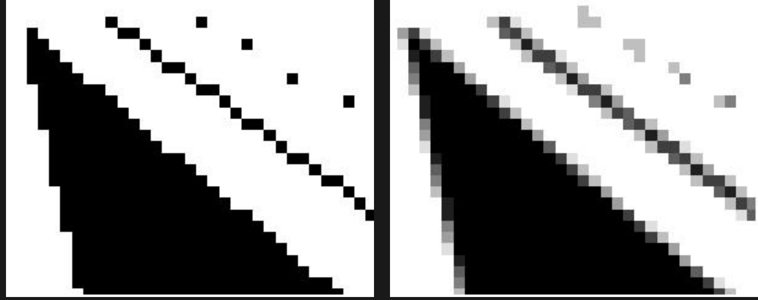
Slide by Tomas Akenine-Möller

Screen-space Anti-Aliasing

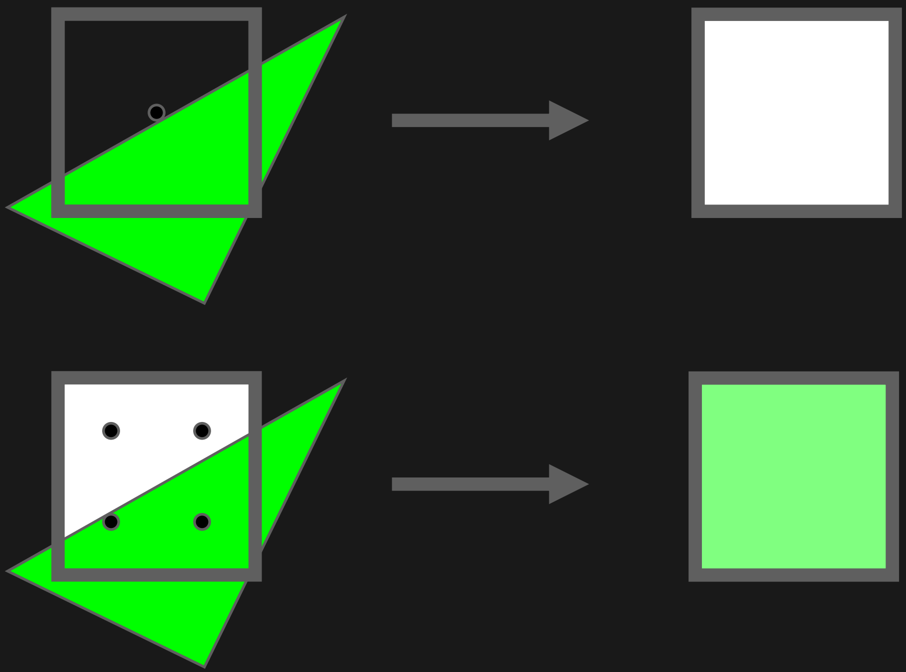
- For better image quality, more sampling per pixel is needed
- For real-time graphics, multi-sampling AA (MSAA) is often used
- [Naiman1998] showed that near-horizontal/vertical edges are in most need of improvement for humans

Slide by Tomas Akenine-Möller

Screen-space Anti-Aliasing



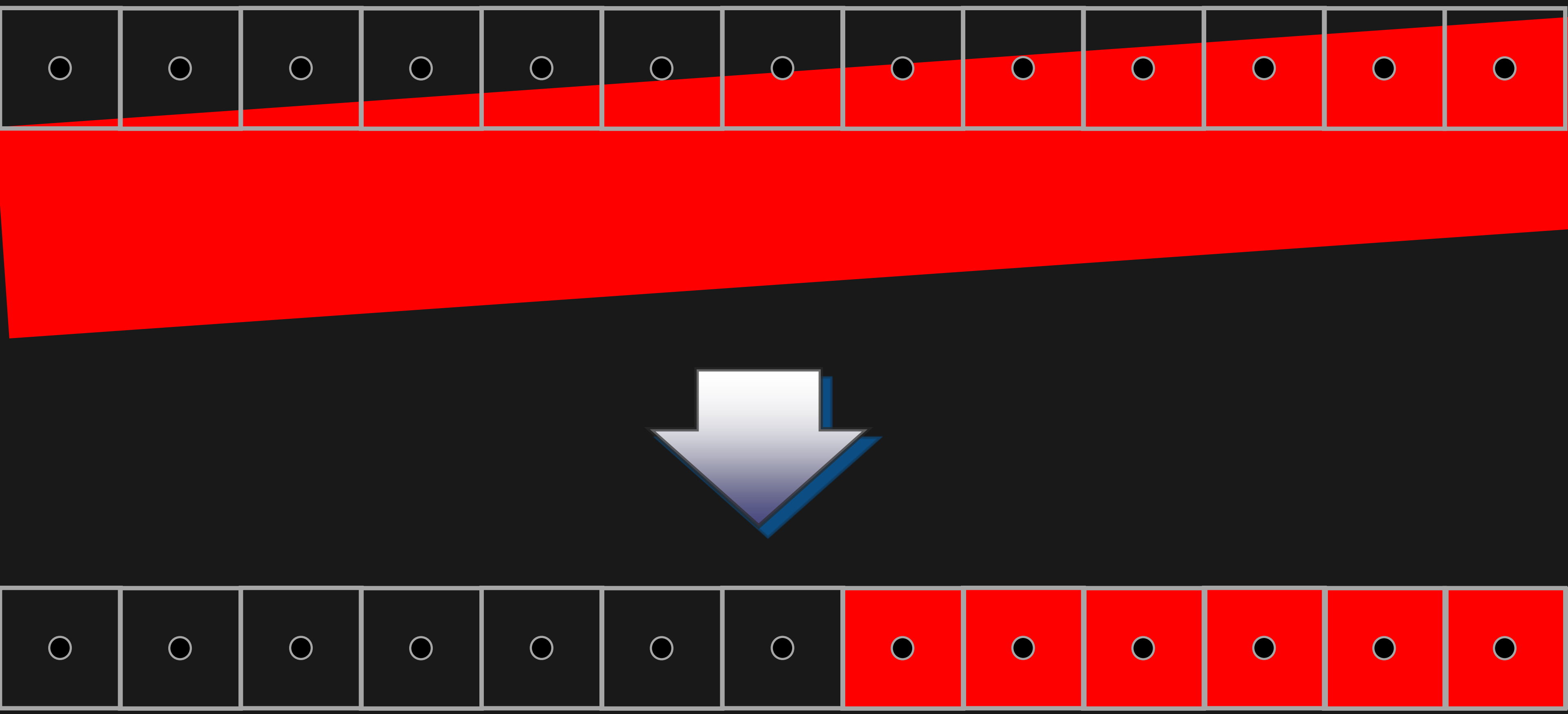
- One sample per pixel is not enough
- Hard case:
  - An edge has infinite frequency content
  - Means no sample rate can fix this for us...
- Multi/Supersampling techniques: use more samples



NOTE: frame buffer needs to be 4x as big!

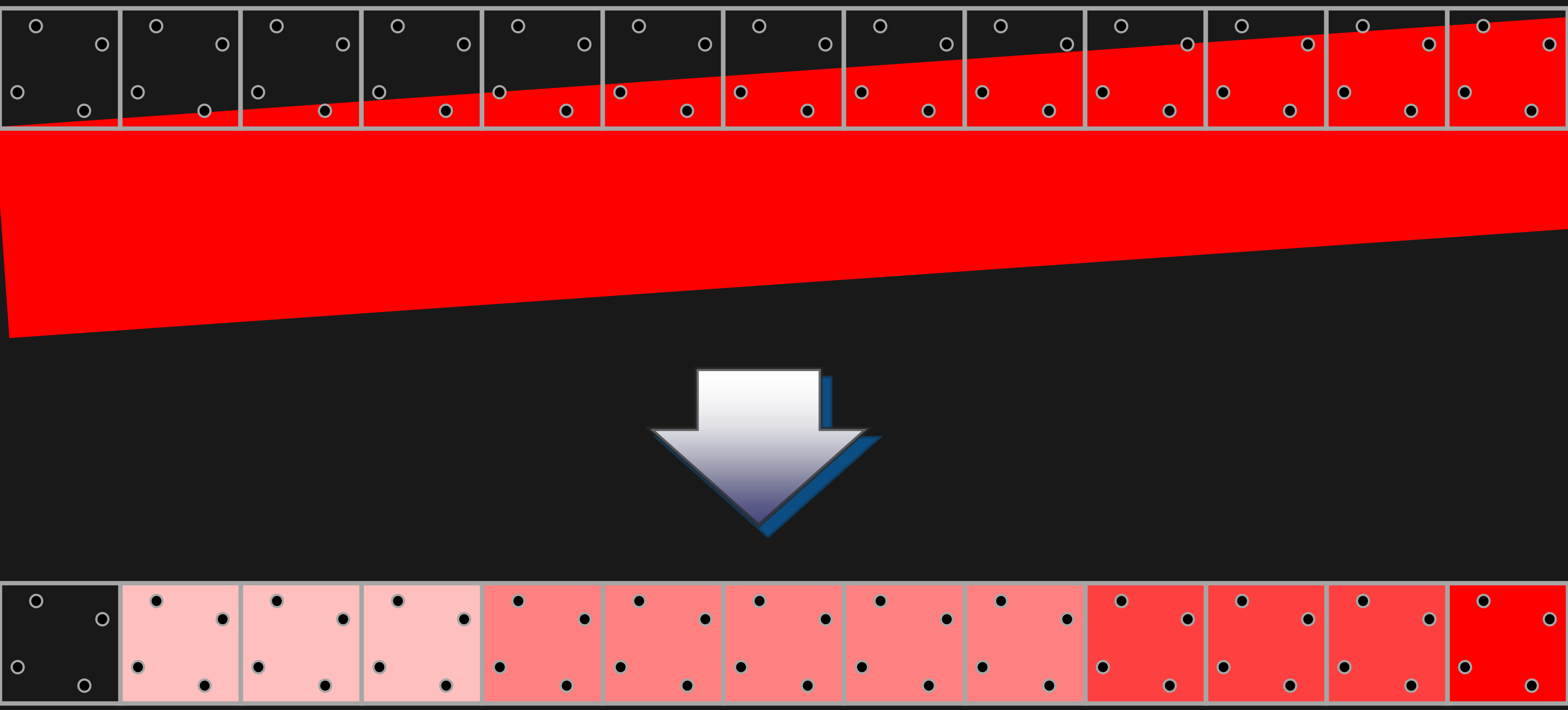
Slide by Tomas Akenine-Möller

A single sample per pixel



Slide by Tomas Akenine-Möller

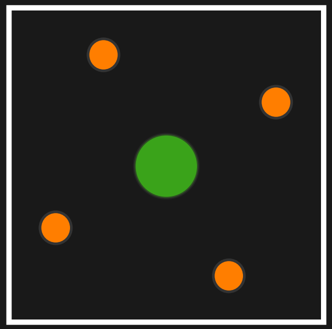
4 samples per pixel
Rotated Grid Supersampling (RGSS)



Slide by Tomas Akenine-Möller

Multi-Sampling Anti-Aliasing (MSAA)

- Observation: the most important thing to anti-alias are the *edges*, and not pixel shading
  - Plus: pixel shading is expensive
- The MSAA approach:
  - Increase geometrical sampling
    - Sample inside/outside triangle several times per pixel
  - But sample pixel shading only *once*



Sample pixel shading in the middle

Sample inside/outside triangle several times

4x MSAA required by DX10.1

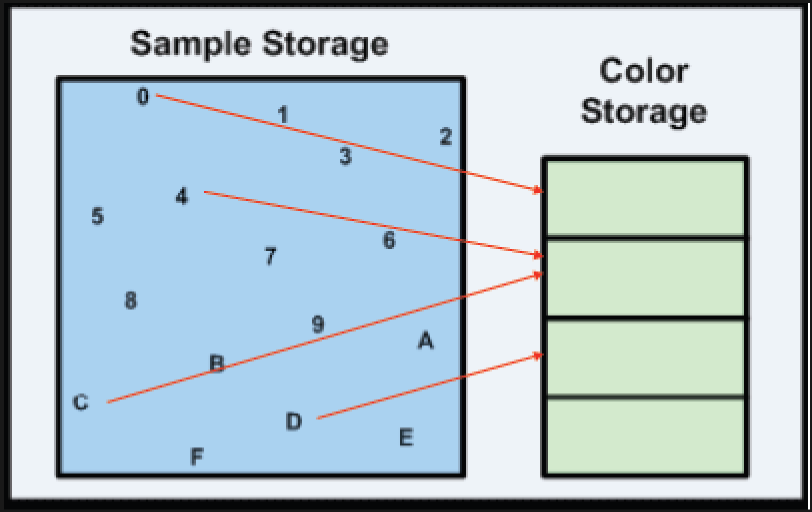
Slide by Tomas Akenine-Möller

Take another look at those images...

- It is really the edges that are in most need of improvement...
- MSAA handles that quite well

Coverage Sample Anti-Aliasing (CSAA)

- "Coverage" means inside or outside a triangle
- Decouples coverage from color/Z/stencil
  - Higher sampling rate for coverage than color
- Per-pixel has a palette of colors
  - Each sample picks a color from a palette

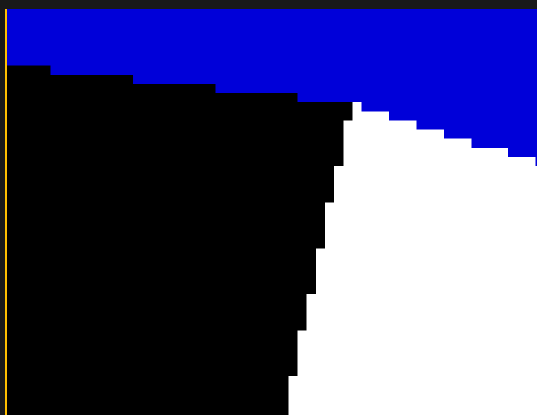


Coverage Sampling AntiAliasing (CSAA)

- Pros
  - Lower memory bandwidth usage
  - Low performance overhead
    - Only additional rasterization tests needed
    - Do not need to Z/Stencil test per coverage sample
- Cons
  - Incorrect if > 4 surfaces are visible through a pixel
  - A form of lossy compression

Slide by Tomas Akenine-Möller

CSAA vs MSAA



No AA

4x MSAA

CSAA
16x coverage
4x color

Slide by Tomas Akenine-Möller

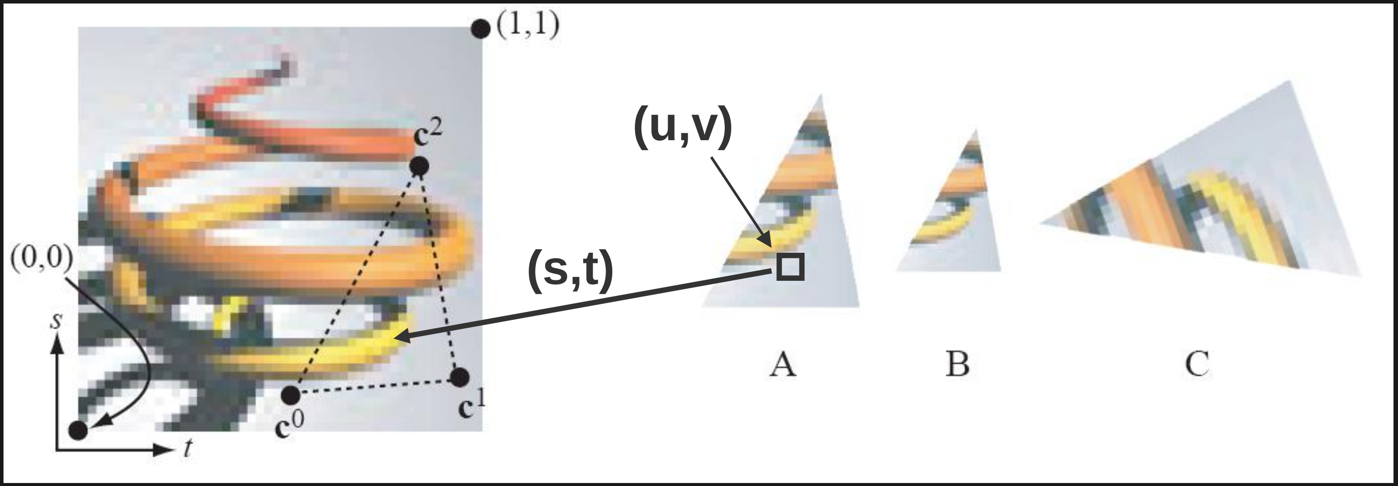
Texturing

Texturing

Image from "Lpics" paper by Pellacini et al.
SIGGRAPH 2005, Pixar Animation Studios

- Surprisingly simple technique
  - Extremely powerful, especially with programmable shaders
  - Simplest form: "glue" images onto surfaces (or lines or points)

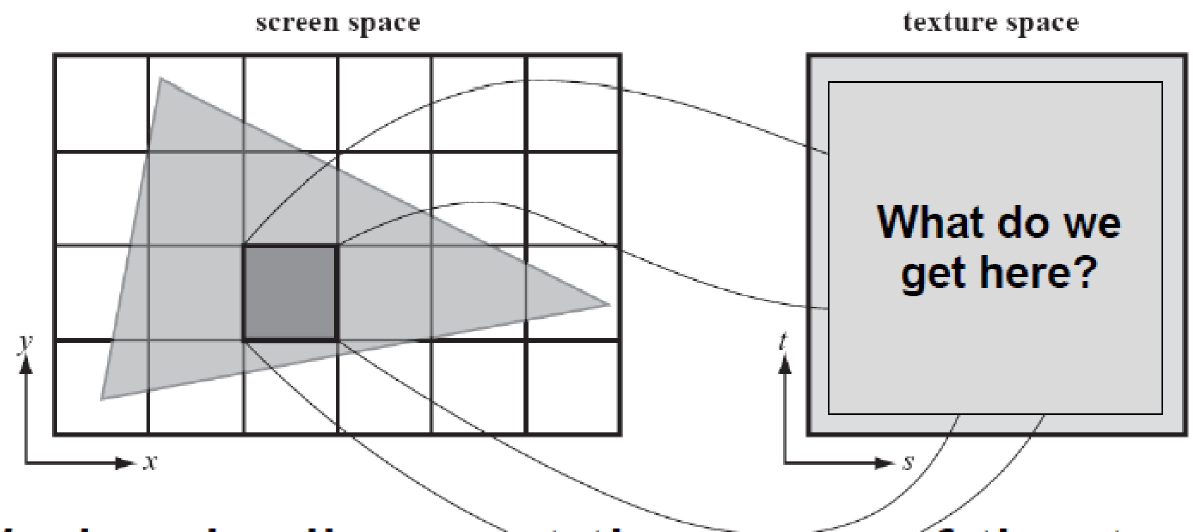
Texture space, (s,t)



- Texture resolution, often 2^a x 2^b texels
- The $\mathbf{c}^k$ are *texture coordinates*, and belong to a triangle's vertices
- When rasterizing a triangle, we get (u,v) interpolation parameters for each pixel (x,y):
  - Thus the texture coords at (x,y) are:

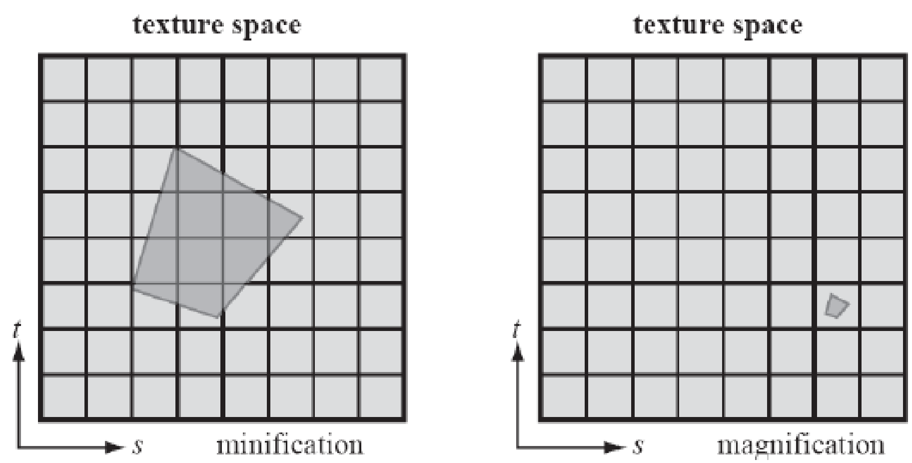
$$(s,t) = (1-u-v)\mathbf{c}^0 + u\mathbf{c}^1 + v\mathbf{c}^2$$

Texture filtering



- We basically want the sum of the texels in the footprint (dark gray) to the right

MINIFICATION

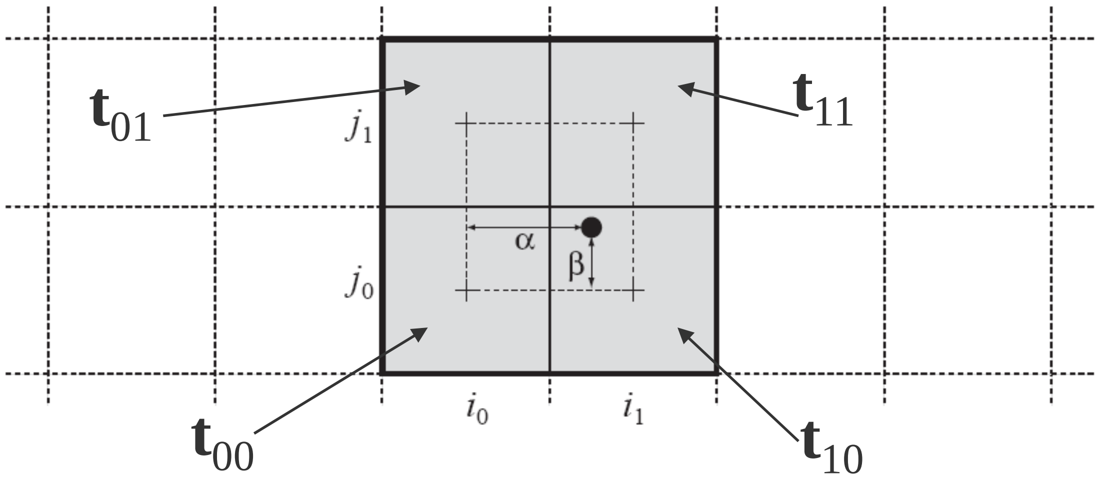


MAGNIFICATION

Texture magnification (1)

- Middle: nearest neighbor – just pick nearest texel
- Right: bilinear filtering: use the four closest texels, and weight them according to actual sampling point

Texture magnification (2)



- Bilinear filtering is simply, linear filtering in x:

$$\begin{aligned} \mathbf{a} &= (1-\alpha)\mathbf{t}_{00} + \alpha\mathbf{t}_{10} \\ \mathbf{b} &= (1-\alpha)\mathbf{t}_{01} + \alpha\mathbf{t}_{11} \end{aligned}$$

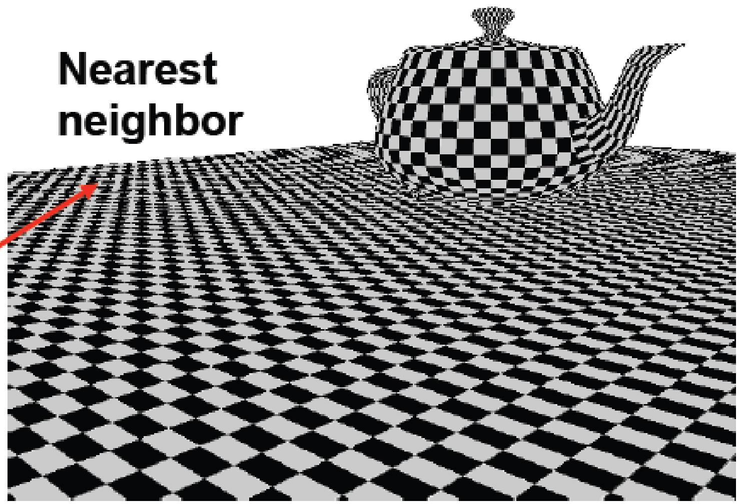
- Followed by linear filtering in y:

$$\mathbf{f} = (1-\beta)\mathbf{a} + \beta\mathbf{b}$$

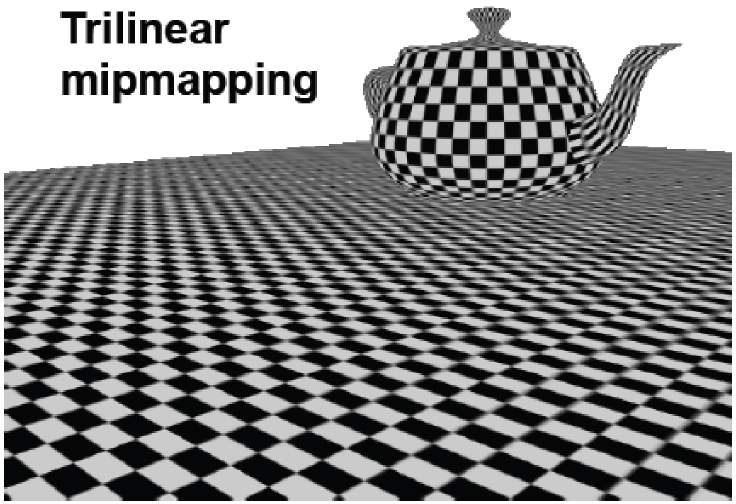
Texture minification

- If nearest neighbor or bilinear filtering is used, then serious flickering will result

Nearest neighbor

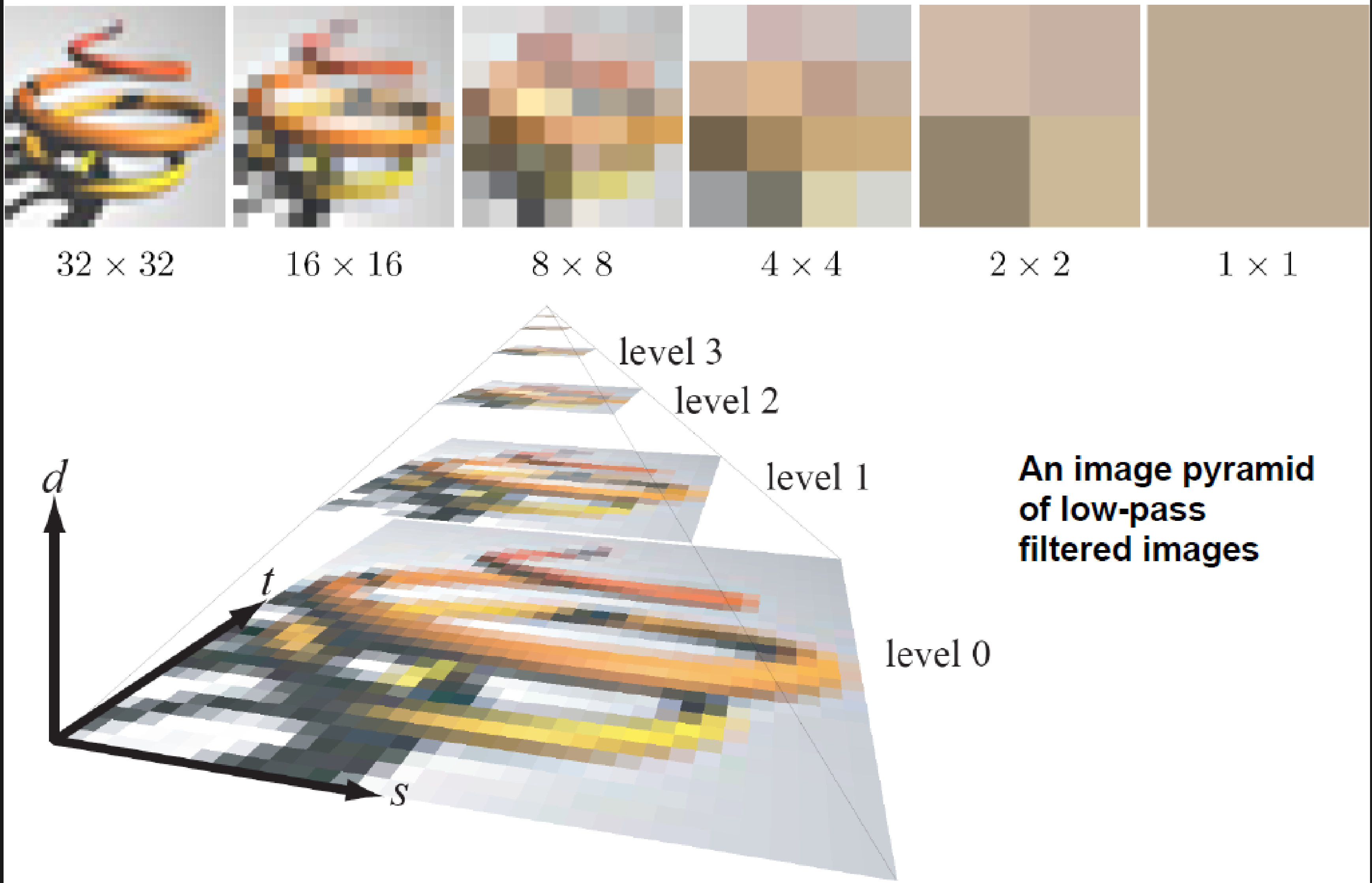


Trilinear mipmapping



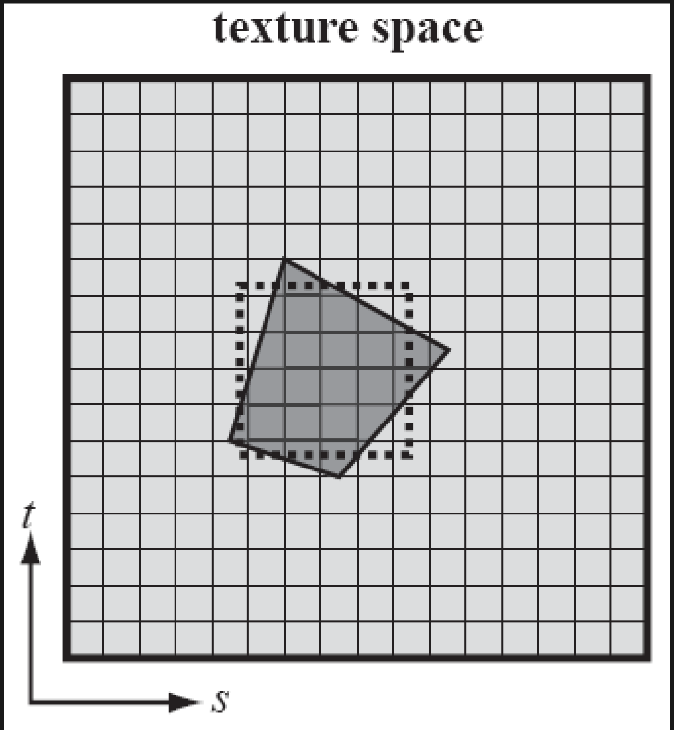
For a pixel here, there is a 50% Change of getting a black texel

Texture minification: mipmapping



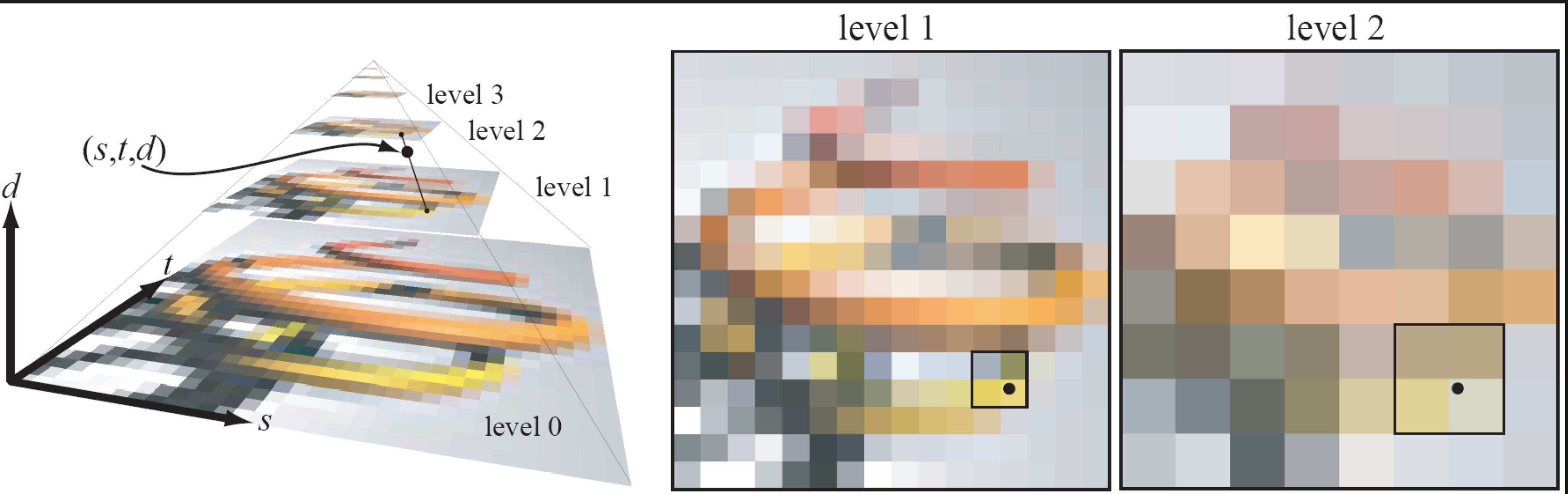
An image pyramid of low-pass filtered images

Trilinear Mipmapping (1)



- Basic idea:
  - Approximate (dark gray footprint) with square
  - Then we can use texels in mipmap pyramid

Trilinear mipmapping (2)



- Compute d, and then use two closest mipmap levels
  - In example above, level 1 & 2
- Bilinear filtering in each level, and then linear blend between these colors → trilinear interpolation
- Nice bonus: makes for much better texture cache usage

Wrap-Up

- The 2003 real-time rendering pipeline supports
  - Programmable vertex shaders (100s of instructions)
  - Programmable fragments shaders (100s of instructions with limited control flow)
- This is the technology in the current game console generations (Microsoft XBox360 and Sony PlayStation 3)
- These APIs spurred a large amount of innovation in rendering and GPU computing (it was the generation before GPU compute languages)

Questions?

- Next lecture:
  - The 2011 real-time rendering pipeline (DirectX11)

Some references

Books

Foley / van Dam – Computer graphics: Principles and Practices / Introduction to computer graphics

"Real-Time Rendering" textbook by Tomas Akenine-Moller, Haines, and Hoffman

Courses

- Kurt Akeley's Stanford course slides, co-founder of SGI, from which this lecture borrows liberally
  - http://graphics.stanford.edu/courses/
- The University of North Carolina at Chapel Hill has another superb department of graphical research and well documented Introduction to Graphics from which this lecture also shamelessly borrows
  - http://www.cs.unc.edu/~pmerrell/comp575.htm
- Illinois also has a graphical programming course taught by experts, though material is heavily infiltrated by NVidia marketing and CUDA (general purpose/non-graphical GPU) models
  - http://courses.ece.illinois.edu/ece498/al/Syllabus.html

Conferences – ACM SIGGRAPH, High Performance Graphics, GDC, EUROGRAPHICS

Chris Thomas' Java 3D applets for beginning graphic programmers (cool switches to play with on basic concepts with immediate visual feedback) http://ctho.ath.cx/toys/3d.html

References 1

Vertices and Primitives

[1] Kurt Akeley, "Open GL", Stanford, Lecture 2, http://graphics.stanford.edu/courses/cs248-07/

[2] David Blythe, "Direct3D 10", SIGGRAPH 2006, http://www.cs.umbc.edu/~olano/s2006c03/ch02.pdf

[3] Kurty Akeley, Pat Hanrahan, "The Graphics Pipeline", Stanford, Lecture 2 http://graphics.stanford.edu/cs448-07-spring/

Transformations

- Kurt Akeley, "2-D Transformations", Stanford, Lecture 7 http://graphics.stanford.edu/courses/cs248-07/
- Kurt Akeley "3-D Transformations", Stanford, Lecture 8 http://graphics.stanford.edu/courses/cs248-07/
- MSDN, "3-D Transformations Overview", Microsoft http://msdn.microsoft.com/en-us/library/ms753347.aspx

Z-Buffering

- Kurt Akeley, "Z-Buffer", Stanford, Lecture 13 http://graphics.stanford.edu/courses/cs248-07/
- Steven Molnar, "Combining Z-buffer Engines for Higher-Speed Rendering", EUROGRAPHIC 1988, pp171-182, http://www.cs.unc.edu/~molnar/Papers/ZComp-hwws88.pdf
- Stewart, Leach, John, "An Improved Z-Buffer CSG Rendering Algorithm", EUROGRAPH/SIGGRAPH 1998 http://www.nigels.com/research/egsggh98.pdf
- Greene, Kass, Miller, "Hierarchical Z-Buffer Visibility", SIGGRAPH 1993 www.cs.princeton.edu/courses/archive/spring01/cs598b/papers/greene93.pdf

References 2

Rasterization

[4] Kurt Akeley, "Rasterization", Stanford, Lecture 5 http://graphics.stanford.edu/courses/cs248-07

- Fredo Durand and Barb Cutler, "Rasterization", MIT, EECS 6.837 Lecture 15, http://groups.csail.mit.edu/graphics/classes/6.837/F03/lectures/15%20Raster.pdf
- Crisu, Cotofana, Vassiliadis, Liuha, "Efficient hardware for Tile-Based Rasterization" ProRISC 2004, http://ce.et.tudelft.nl/publicationfiles/964_12_crisu_prorisc2004.pdf

Textures

[5] Kurt Akeley, "Texture Mapping", Stanford, Lecture 10 http://graphics.stanford.edu/courses/cs248-07

[6] Paul Merrell, "Texture Synthesis", Nov 6, 2008 http://www.cs.unc.edu/~pmerrell/comp575.htm

- Eckstein, Surazhsky, Gotsman, "Texture Mapping with Hard Constraints", EUROGRAPHICS 2001, http://www.cs.technion.ac.il/~gotsman/AmendedPubl/TextureMapping/TextureMapping.pdf
- "Texture Mapping", SIGGRAPH.org, http://www.siggraph.org/education/materials/HyperGraph/mapping/texture0.htm
- Daniel Pinkwater, "The Hoboken Chicken Emergency" http://www.amazon.com/Hoboken-Chicken-Emergency-Daniel-Pinkwater/dp/0689828896

References 3

Anti-Aliasing

[7] Kurt Akeley, "Multisample Antialiasing", Stanford, Lecture 6 http://graphics.stanford.edu/courses/cs248-07

- Kurt Akeley, "Pre-Filter Antialiasing", Stanford, Lecture 4 http://graphics.stanford.edu/courses/cs248-07
- Kurt Akeley, "Sampling and Aliasing", Stanford, Lecture 3 http://graphics.stanford.edu/courses/cs248-07
- Open GL multisample specs http://www.opengl.org/registry/specs/ARB/multisample.txt
  http://www.opengl.org/registry/specs/SGIS/multisample.txt
- "AntiAliasing Techniques" SIGGRAPH.org, http://www.siggraph.org/education/materials/HyperGraph/aliasing/alias0.htm

Lighting

[8] Paul Merrell, "Shading", Sept 25, 2008 http://www.cs.unc.edu/~pmerrell/comp575.htm

[9] Kurt Akeley, "Illumination and Direct Reflection", Stanford, Lecture 12 http://graphics.stanford.edu/courses/cs248-07

- Stokes, Ferwerda, Walter, Greeenberg, "Perceptual Illumination Components: A New Approach to Efficient, high Quality Global Illumination Rendering", SIGGRAPH 2004, http://www.cis.rit.edu/jaf/publications/PICS_SIGGRAPH2004.pdf
- Cabral, Orlano, Nemen, "Reflection Space Image-Based Rendering", SIGGRAPH 1999
  https://eprints.kfupm.edu.sa/61732/1/61732.pdf

References 4

Low-Level APIs

– MSDN, “HLSL”, Microsoft Corp, http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx
– OpenGL, “OpenGL & OpenGL Utility Specifications”, OpenGL.org http://www.opengl.org/documentation/specs/
– Khronos, “OpenCL”, Khronos Group http://www.khronos.org/opencl/
– R600-Technology, “R600-Family Instruction Set Architecture”, AMD corp, http://www.x.org/docs/AMD/r600isa.pdf

ATI Hardware

[10] Anadtech, “The Radeon HD 4850 & 4870”, http://www.anandtech.com/video/showdoc.aspx?i=3341&p=1
– ATI developer publications http://ati.amd.com/developer/techreports.html
– AMD main site http://www.amd.com/us-en/
– Beyond3D, “Sir Eric Demers on AMDR600,” http://www.beyond3d.com/content/interviews/39/
– R.K. Montoye, E. Hokenek and S.L. Runyon, “Design of the IBM RISC System/6000 floating-point execution unit,” *IBM Journal of Research & Development*, Vol. 34, pp. 59-70, 1990.
– Pics etc may be google’d with ease