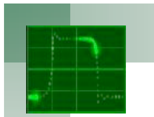
Graphics and Genres

CMPUT 299

Finnegan Southey

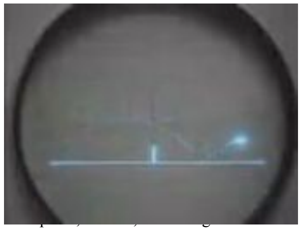
XBox Live GamerTag: Alea



The Early Days

- The very earliest video games (pre-1975) were custom built machines.
- Designed/built by engineers (like a TV).

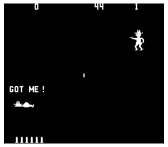
Tennis for Two
(Brookhaven Labs,1958)



Pong (Atari, 1972)

Hardware vs. Software

- Later games (1975 to present) were built using new-fangled *microprocessors*
  - General-purpose *hardware* that runs *software*.
  - No need to engineer from scratch for every game.



Gunfight (Taito, 1975)

Development of Early Games

- Programs written by one individual.
- Graphics, sound, controls, rules, AI…

…all by one person.

Development of Early Games

- Games were simple.
- The machines were still quite simple
  - Very limited storage
    - no “pictures” or recorded music
  - Limited speed
    - focus on moving small things around on the screen
- Only so much one could do
  - more people would be a waste of effort

State vs. Dynamics

- Fundamental distinction in computing (and many other things)
  - State
    - All information that describes the game at a given moment
  - Dynamics
    - The way one state turns into another

State

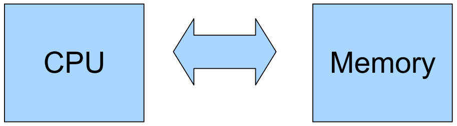
- positions of all game entities
- walls
- resources: health, magic points, money, fuel, etc.
- points: score, tokens collected, goals achieved
- inventories: weapons, magic items, gadgets, food
- switches: doors locked/unlocked, levers pulled
- velocities of game entities
- character names, stats, description
- much, much more…

Dynamics

- agents moving due to player controls
- agents moving due to artificial intelligence
- agents moving due to “physics”
- agents not moving due to “collisions”
- updating view of the world
- health changes from damage/healing
- special powers/action executed
- everything that changes one state into another…

Memory and Processors

- Two main parts of a computer
  - Memory stores state
  - Processors access and change memory (dynamics)
- The main processor in a computer is called the *central processing unit* (CPU).



Memory Size

- Memories have a size
  - in the 70’s, measured in kB (kilobytes)
  - in the late 80’s and 90’s, measured in MB (megabytes) (1 MB = 1000 kB)
  - late 90’s to present day, measure in GB (gigabytes) (1 GB = 1000 MB)
- Early games had 4 kB
- Games today will cheerfully use several GB
  - a factor of one million (1 DVD holds over 4 GB)

Processors

- Processors are rated by how quickly they can perform calculations on things in memory
- A *program* is a set of instructions that tells the processor what to do with the information in memory.
- Programmers write these instructions.
- Early microprocessor games were a program + a tiny amount of data
- A contemporary game is a program + lots and lots of data

Memory Types

- Memories can be
  - fast or slow
    - memory for showing graphics and doing calculations is fast (it can be accessed quickly)
    - harddisks and DVDs are much slower but hold much more
  - read-only or read/write
    - read-only memory (like a game’s DVD) cannot be changed
    - read/write memory can be changed

Memory Types

- Another distinction
  - volatile: erased when power turned off
  - non-volatile: retains information unpowered
- And another
  - moving parts vs. no moving parts
    - can affect reliability/portability

Storage for Games

- RAM chips: volatile, read-write
  - microchips for running game
  - part of game machine
- ROM chips: non-volatile, read-only
  - expensive per amount of memory
  - microchips

Storage for Games

- cartridges
  - ROM chips in a plastic case
  - connectors attach chip to game machine
  - durable
- cartridges with save memory
  - ROM chips + non-volatile, read/write memory chips
  - allows storage of saved games/high scores
- memory cards
  - small cartridges with non-volatile, read/write memory chips
  - used to save games when cartridges went out of style

Storage for Games

- cassette (magnetic tape): non-volatile, read-write
  - moving parts
  - early personal computers (and some arcade?)
  - not very durable
  - sequential access (have to rewind/fast forward to reach different information)
- floppy disk: non-volatile, read-write
  - moving parts
  - early personal computers
  - not very durable
  - “random” access (can quickly access any piece of information)

Storage for Games

- hard-disk: non-volatile, read-write
  - moving parts
  - later personal computers and XBox
  - much larger storage and faster access than floppy
  - expensive
- laser disc: non-volatile, read-only
  - moving parts, tons of storage
  - able to store full screen video
  - expensive (videophile technology)
  - arcade (Dragon's Lair, Mach 3, Space Ace)

Storage for Games

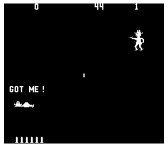
- magnetic strip cards: non-volatile, read-write
  - arcade for saving stats/games (Initial D, F-Zero AX, Tekken 5)
  - very limited storage
  - very cheap
- CD-ROM: non-volatile, read-only
  - moving parts, lots of storage
  - cheap to mass produce
  - arcade (Killer Instinct) and later personal computers
  - consoles (3DO, Saturn, Playstation) (N64 still cartridge!)

Storage for Games

- DVD-ROM: non-volatile, read-only
  - moving parts, tons of storage
  - cheap to mass produce
  - personal computers
  - consoles (XBox, Playstation2)
- Custom disc formats (Dreamcast, Gamecube)
- Coming up: DVD-HD and BluRay (PS3)
  - bigger DVD
- Piracy! (arrrr…)

Software-Based Games

- Once microprocessors were used for games, *programmers* took control.
- The advent of personal computers (~1976) opened up the field to "amateurs".
- Basement game programmers



Gunfight (Taito, 1975)

Cathode Ray Tubes (CRT)

- Braun (1897) - CRT oscilloscope
- Zworykin (1929) kinescope (early TV)
- Du Mont (1931) first commercial TV tube

Tennis for Two

(Brookhaven Labs,1958)

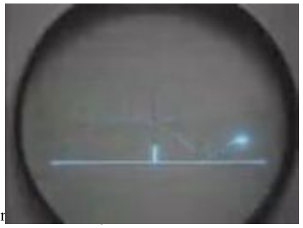


Diagram of CRT

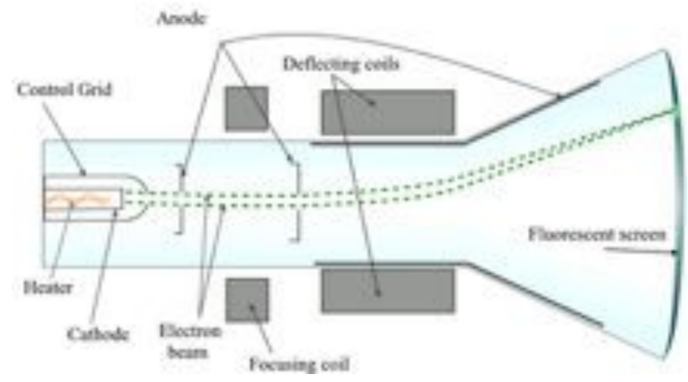
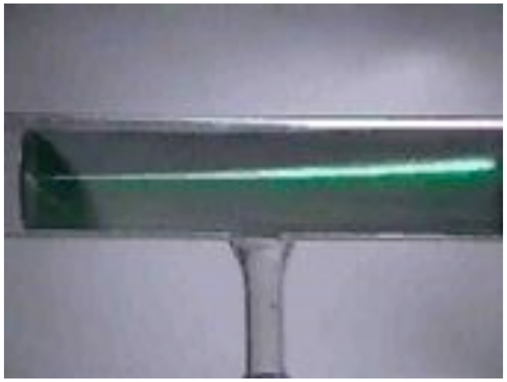


Diagram courtesy of Wikipedia

Very pretty, but where's the gamepad?



Video courtesy of the University of Illinois, Dept. of Chemistry

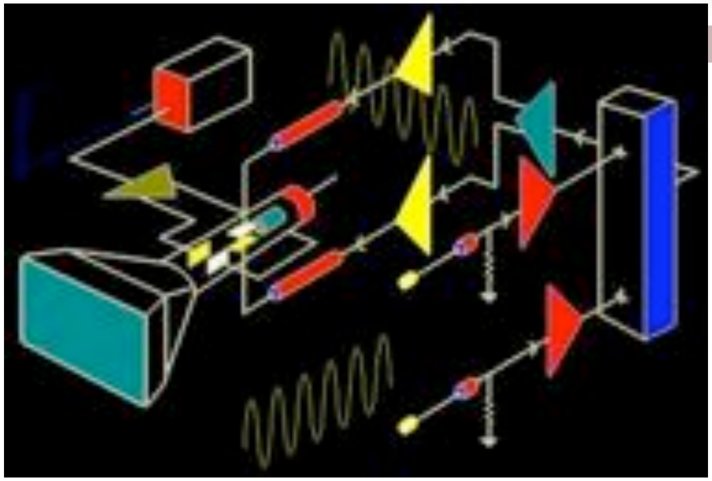


Diagram courtesy of Williamson Labs

Vector Graphics

- Use CRT's beam like a pen.
- Turn beam on and off to draw lines.
- Use magnets to guide it.
- How many lines you can draw depends on
  - how fast you can move the beam around
  - how long the image stays on the fluorescent screen
- Draw all lines for one *frame*… repeat.

Games with Vector Graphics (no more after 1985)

- Asteroids (Atari, 1979)
- Lunar Lander (Atari, 1979)
- Battlezone (Atari, 1980)
- Red Baron (Atari, 1980)
- Tempest (Atari, 1980)
- Space Fury (Sega, 1981)
- Star Wars (Atari, 1983)

Issues with Vector Graphics

- Time to draw frame depends on complexity of frame
- Beam moves in arbitrary pattern
  - different patterns possible for same picture
  - what's the best (fastest) pattern?
- Directly control beam with game

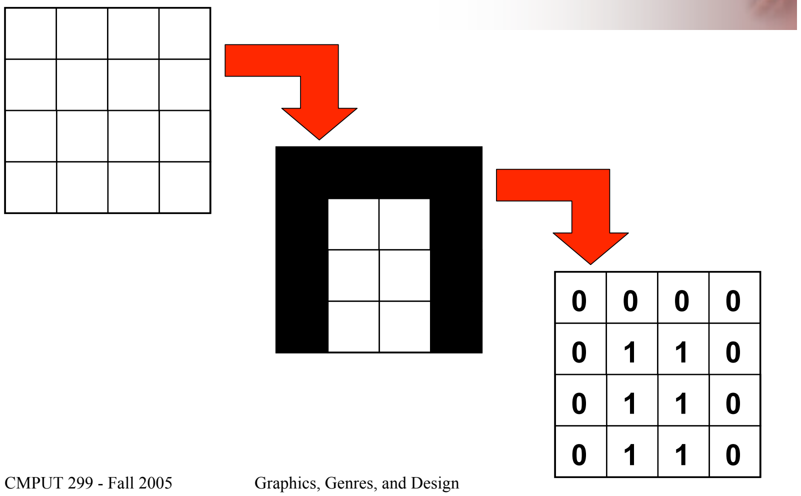
Cool Things About Vector Graphics

- Wireframe (3D)
- Smooth lines (even diagonals)
- It's all green and glowy and stuff…
- Makes me feel like I'm in a submarine or something…

Raster Graphics

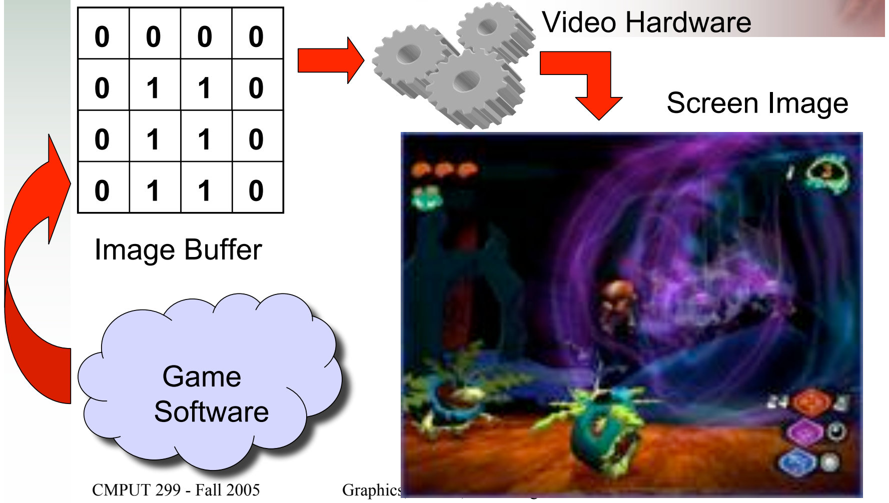
- Basis of TV and almost all graphics today
- Treat screen like a “grid”
- Move beam in fixed pattern lighting up the screen in little dots
- These dots are called *pixels*
- Change beam intensity to make pixels brighter or darker

Raster (aka Bitmap) Images



0	0	0	0
0	1	1	0
0	1	1	0
0	1	1	0

Rendering the Screen



0	0	0	0
0	1	1	0
0	1	1	0
0	1	1	0

Image Buffer

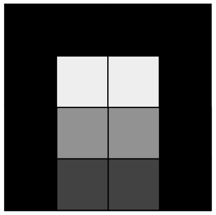
Video Hardware

Screen Image

Game Software

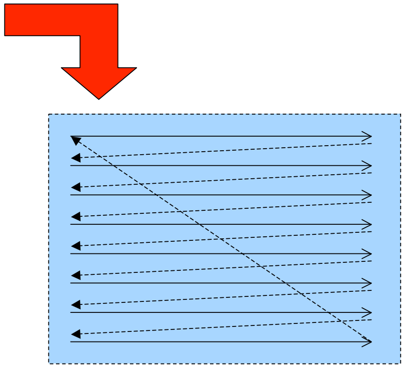
Grayscale

0	0	0	0
0	100	100	0
0	50	50	0
0	10	10	0

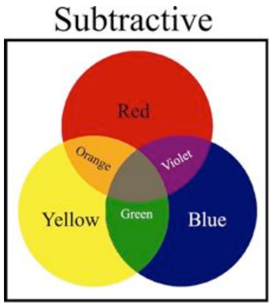


CRT Screen Refresh

0	0	0	0
0	1	1	0
0	1	1	0
0	1	1	0



Colour Pixels



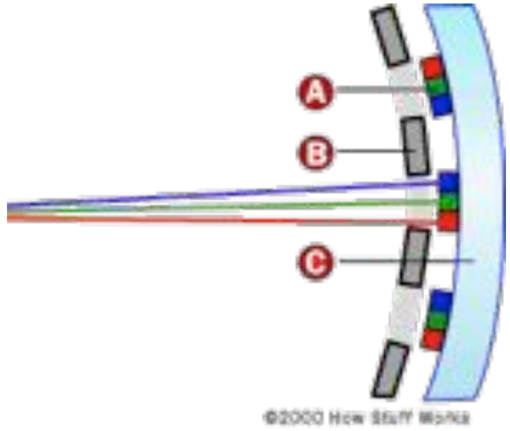
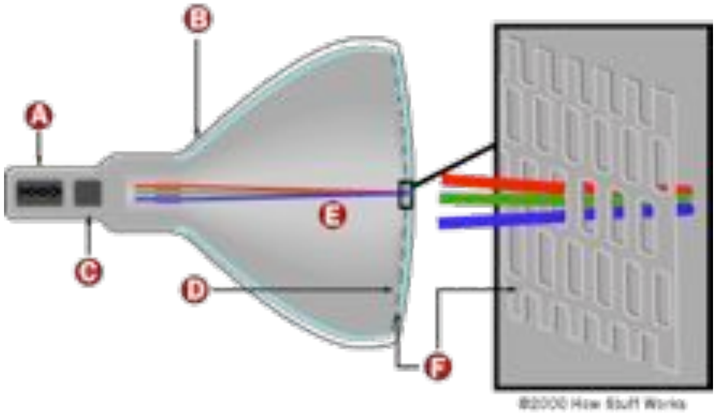


Diagram of Colour CRT

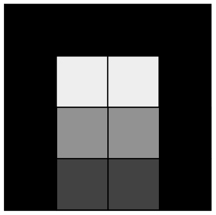


RGB Images

0	0	0	0
0	100	100	0
0	50	50	0
0	10	10	0

0	0	0	0
0	100	100	0
0	50	50	0
0	10	10	0

0	0	0	0
0	100	100	0
0	50	50	0
0	10	10	0



How many pixels?

- If we make pixels smaller, we can fit more!
- Fineness of grid called *resolution* (width x height)
- Typical television resolution
  - grid of 648 x 486 (~300,000 pixels)
- Computer screens, HDTV
  - 1024 x 768 (~780,000 pixels)
  - 1280 x 1024 (~1.3 million pixels)
  - 1600 x 1200 (~1.9 million pixels)
- Old games
  - Space Invaders - 224 x 240 (53,760 pixels)

How many colours?

- Suppose we're using grayscale.
- If we have 10 different values for brightness, then we get black, white, and 8 shades of gray.
- If we have 10 different values for each of red, green, and blue, then we have
  - 10 x 10 x 10 colours (1000)

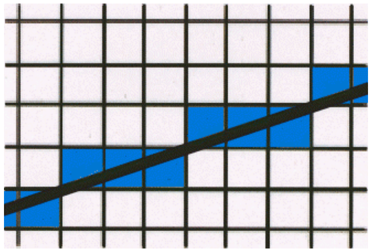
Colour Depth

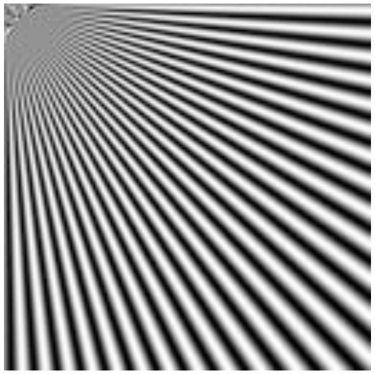
- The number of values we have for colours is often called *colour depth* and is usually measured in *bits* (k bits = 2^k values)
- 4 bits = 16 x 16 x 16 = 4096 colours
- 8 bits = 256 x 256 x 256 = ~ 16m colours
- 24 bits = 16m x 16m x 16 m = ~4700000000000000000000
- 32 bits = 4b x 4b x 4b = 7.9 x 10^28

Framerate

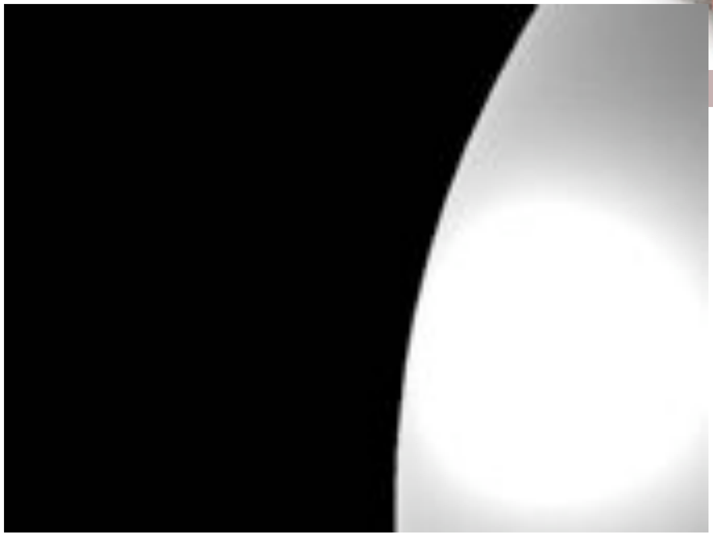
- How long does it take to "draw" the next frame?
- If it takes a tenth of a second
  - we can draw 10 frames per second
- Common *framerates*
  - Minimum for "smooth" motion ~ 15 fps
  - Film ~ 24 fps
  - TV ~ 30 fps
  - Games
    - RTS ~ 8 fps
    - platformers ~ 30 fps
    - first person shooters ~60 fps

Raster Side Effect: Aliasing





a.k.a “the jaggies”



Early Raster Games

- Space Invaders (Taito, 1978)
- Super Breakout (Atari, 1978)
- Galaxians (Midway, 1979)
- Pacman (Namco/Midway), 1980)
- Berzerk (Stern, 1980)
- Centipede (Atari, 1980)
- Defender (Williams, 1980)
- Cheeky Mouse (Universal Corp., 1980)
- Missile Command (Atari, 1980)
- Warlords (Atari, 1980)
- Donkey Kong (Nintendo, 1981)

Vector vs. Raster

Vector	Raster
clean lines	aliasing
outlines only	filled areas
screen refresh depends on frame	screen refreshes at a fixed rate
direct control	image buffer
special monitors	home TV’s

Vectrex

- Vector did make it to the home!

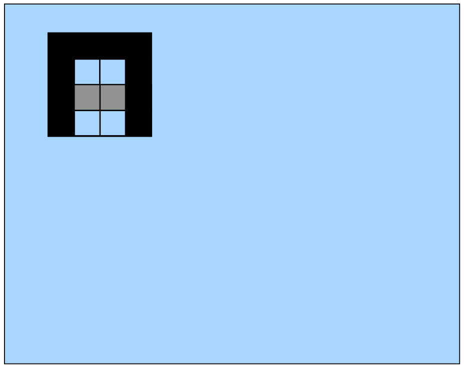
Last Time…

- Computational Model
- Storage for Games
- Graphics
  - CRT
  - Vector Graphics
  - Raster Graphics
  - Bitmaps
  - Resolution, Colour Depth, Frame Rate

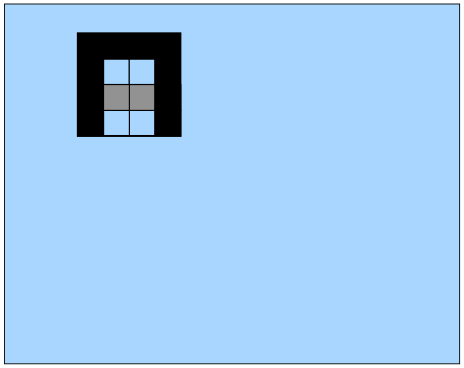
This time…

- Raster wins
- More 2-D graphics
  - Sprites, Collisions, Layers, Scrolling
- Specialization in Hardware
  - Trends in specialization/generalizations
- 3-D Graphics

Background

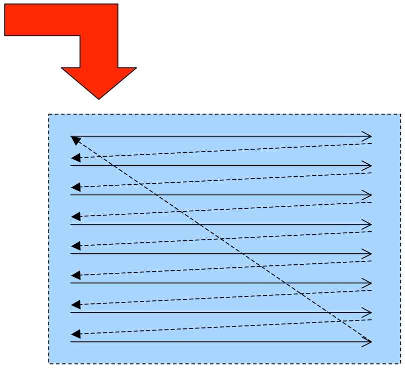


Background

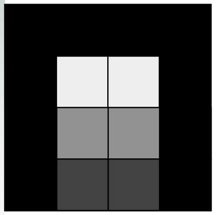


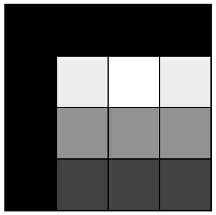
CRT Screen Refresh

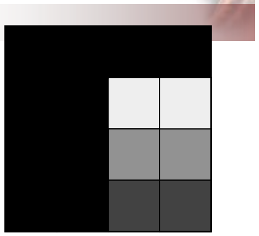
0	0	0	0
0	1	1	0
0	1	1	0
0	1	1	0



Moving Things Around





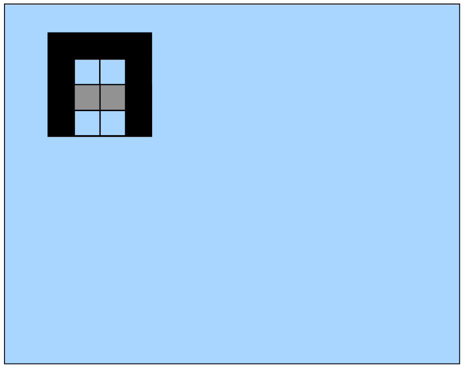


0	0	0	0
0	100	100	0
0	50	50	0
0	10	10	0

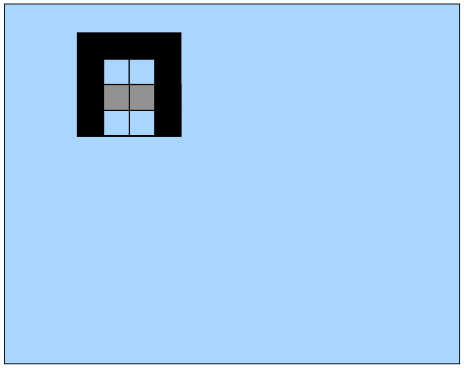
0	0	0	0
0	100	100	100
0	50	50	50
0	10	10	10

0	0	0	0
0	0	100	100
0	0	50	50
0	0	10	10

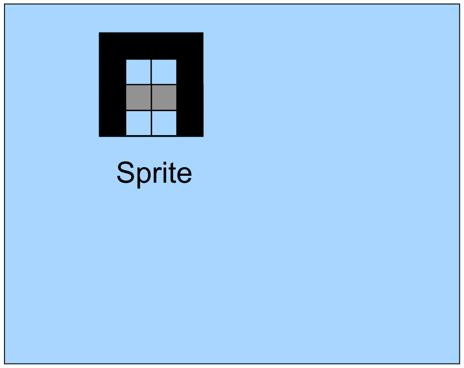
Background



Background



Background

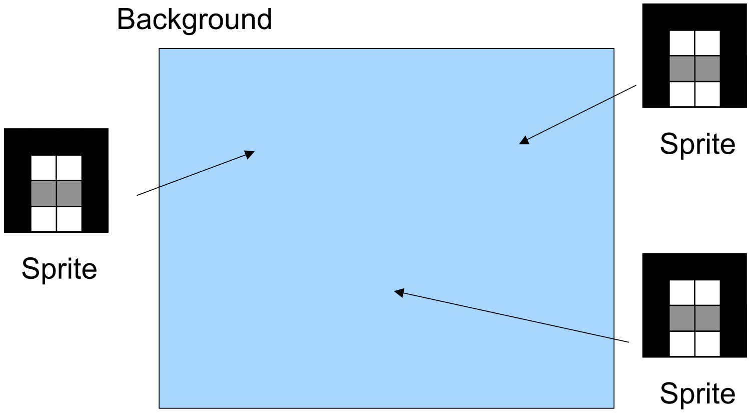


Sprite

Sprites

- Special purpose hardware for moving things around.
- A *sprite* is a small image that can be quickly drawn against a background. It has its own small image buffer.
- Includes *transparency*.
- Hardware sprites somewhere before 1980
  - Pacman (Namco, 1980) – eight hardware sprites
  - 8-bit personal computers (Atari 400/800, Commodore 64)

Background



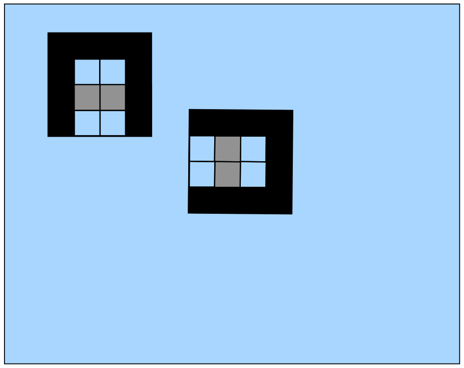
Sprite

Sprite

Sprite

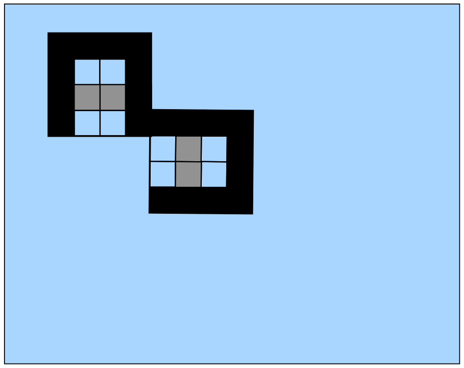
Collisions

Background



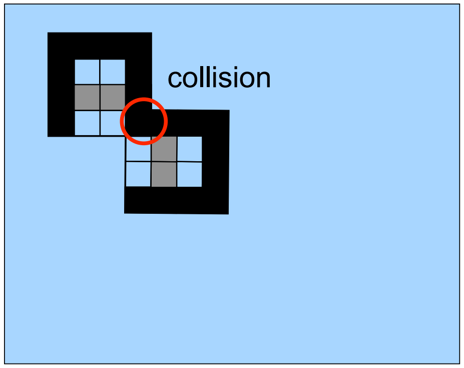
Collisions

Background



Collisions

Background



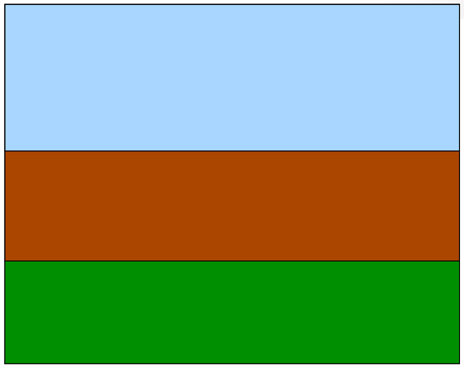
Collisions and Sprites

- Collisions a huge part of determining action in a game.
- Have to check whether moving one graphic causes it to intersect with another.
- Tedious and slow checking of numbers.
- Sprites to the rescue again.
  - Hardware support for *collision-detection*.

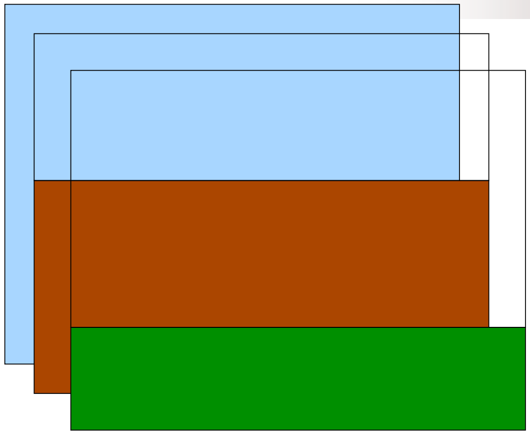
Parallax Scrolling

- Sprites change independent of background
- Now we can change the background at will without worrying about sprites
- Good for motion effects, commonly called *scrolling*
- Can give even better effects with *parallax scrolling*
- Moon Patrol (Irem, 1982)

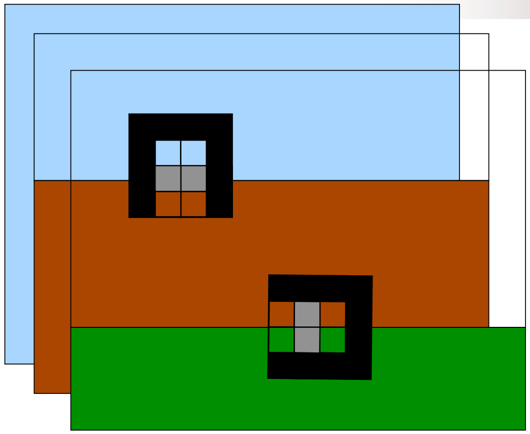
Layers



Layers



Layers



Hardware Support for Scrolling

- Again, special purpose hardware allows multiple backgrounds, all scrolling at different speeds.
- So special 2-D graphics hardware allows
  - sprites
  - multiple, layered scrolling backgrounds
  - collision-detection

Last Generation Sprite Hardware

- Playstation
  - 4,000 8x8 sprites at 60 fps
  - scaling, rotation, transparency, and more
- Saturn
  - around 8,000 sprites (flat-shaded polygons)

Sega fought the wrong battle.

So what?

- Early move from custom built machines to microprocessors.
- Later move to custom hardware for sprites and layered backgrounds.
- A move toward more general-purpose equipment to allow innovation…
- … and a move to specialized equipment to improve performance.

- Have you seen this man?

The Rise of Teams

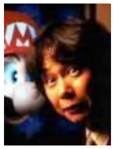
- Better hardware allowed more ambitious games.
- A competitive, fast-moving market increased the number of programmers
- Split the game up into two or three pieces, each programmed (somewhat) separately
- Still required a broad cross-section of skills
  - albeit not necessarily deep skills in many areas

Specialization

- Larger games → Larger teams (4 to 8)
- Stable companies producing multiple games
- Specialization begins
  - someone does graphics
  - someone does sound
  - someone does music
  - someone does gameplay
  - someone does levels
  - etc…

Contemporary Development

- Multiple teams totaling around 100 people
- Communications becomes a real problem
- Some teams rarely (if ever) interact
- Others must interact all the time
- More like a movie production



The Management

- Large projects require stability
  - money and management
- Small publishers giving way to larger publishers with more stable management
- Still a lot of flux
  - teams often put together for a single project

It's all about teams

The Game Designer

- Someone has to coordinate all of these activities to ensure a coherent product.
- Need to understand something of everything.
- The game designer is a *generalist*.

Specialization ↔ Generality

- Industry has gone through many phases and cycles
  - workforce
    - small teams of generalists, specialists, designers as generalists
  - special hardware vs. general purpose
    - graphics, sound, soon physics
  - custom software vs. reusable software
    - reusablility trend recently, partly necessary because games are becoming very expensive (> $50 million for some AAA titles)
    - licensed game engines (Quake, Unreal, Lithtech)
    - Havok physics engine
    - Middleware: Renderware
- game design too (genre refinement & cross-genre)

Content over Container

- Early games didn’t separate *content* from the rest of the game
  - graphics/sound drawn/played directly by the program
- Natural separation into *game engines* and the *data* for different parts of the game
  - level data, graphics textures, music, sound effects, character stats, etc.
- Content is now the largest part of games and the most expensive part.
- Where from? Artists, but increasingly, the real world.

Revolution vs. Evolution

- hardware sprites vs. general purpose graphics
  - many more objects on screen managed by custom hardware
- personal computers vs. arcade
  - opened up development to “amateurs”
- larger storage and recorded music/graphics
  - laser disc (Dragon’s Lair), hard disc (Killer Instinct), and CD (everything now)
- 2-D → 3-D
  - Alone in the Dark, Ultima Underworld, Wolfenstein 3-D, Super Mario 64

3-D Graphics

- Complex subject
- Screens are 2-D
- Produce a 2-D image of a 3-D world
  - Build our 3-D world
  - Point a virtual camera at it
- How do we describe the objects in the world?
- What does our camera do?
- Process of producing a 2-D image is called *rendering*.
- Often call the part of the world we're rendering a *scene.*

Types of Rendering

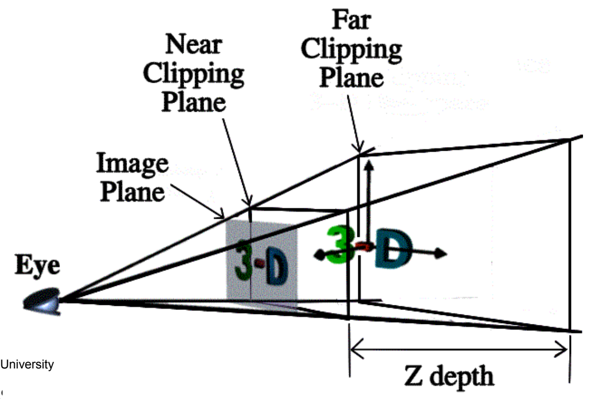
- Non Real-Time (e.g., *raytracing*)
  - compute paths of beams of light in world and figure out their colour when they hit the camera's field of view
  - slow (a single frame may take minutes or hours) but high quality
  - used in movies like Pixar's Toy Story, etc.
  - sometimes used to *pre-render* scenes in games
  - these images can be used as fixed background of high-quality
- Real-Time
  - draw image in real-time (quickly enough to animate)
  - lower quality but world can change with player actions

The Camera

- A camera is a view into the world
- It occupies a position in 3-D space (x,y,z)
- It points in a direction
- It has viewing angles that determine how much it can see right-left and up-down
- May have a maximum depth of view (max distance)
- We can have special effects related to the camera
  - lens flares, fish eye
  - hand-held, out-of-focus, motion blur
- 1st person vs. 3rd person

The Camera: Frustum

- Volume visible to camera: *frustum*



© George Otto, Penn State University

The Camera: Styles

- Fixed cameras
  - finite position and direction (Myst, 7th Guest)
  - fixed position and direction (Resident Evil)
  - fixed position (Myst 3)
  - fixed direction (some RTS, isometric view)
  - fixed path (rail-shooters like Virtua Cop)
  - fixed path with free direction (rail-shooters like Panzer Dragoon)

The Camera: Styles

- Free cameras
  - chase camera (3rd person, follows behind)
  - chase with right/left (3rd person, follows but allows right/left glancing)
  - free-look (1st person, looks wherever you point)
  - look-spring (1st or 3rd camera can be moved but automatically returns to its default when released)
  - rear-view (look behind you)
  - arbitrary view (e.g. side view)
  - droppable (move and then leave it there)
  - switching from 1st to 3rd or vice versa

The Camera: Friend or Foe?

- Obstructions in 3rd person view
- Where does the chase camera go when you back into a wall?
- Camera won’t allow angle you want
- Camera too slow
- Camera too erratic or has jerky transitions
- Camera too automatic or too manual
- Camera really just wants to kill you (any 3-D Sonic game)
- Can follow crazy trajectories, though!

Simple “3-D” Tricks

- Scaling
  - making 2-D sprites (or *bitmaps*) bigger or smaller to simulate moving toward or away
- Isometric view
  - 2-D artistic style showing perspective
- But we want to have a world with real 3-D objects (i.e., they occupy some *volume* of space)

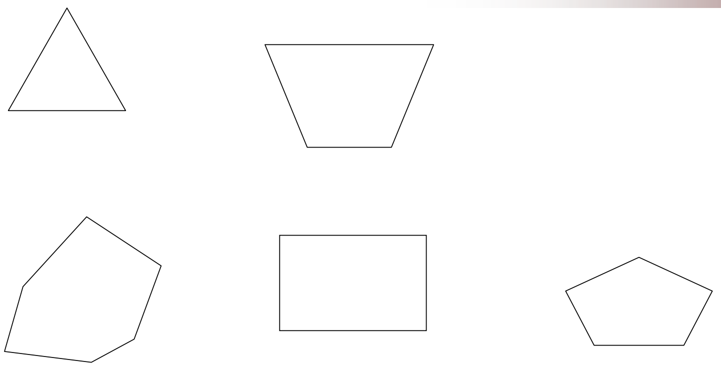
3-D Objects

- Need to represent objects in the world
- Don’t necessarily care what they look like inside
- Let’s make “hollow” objects
- Defined by *surfaces*
- We can build surfaces out of *polygons*

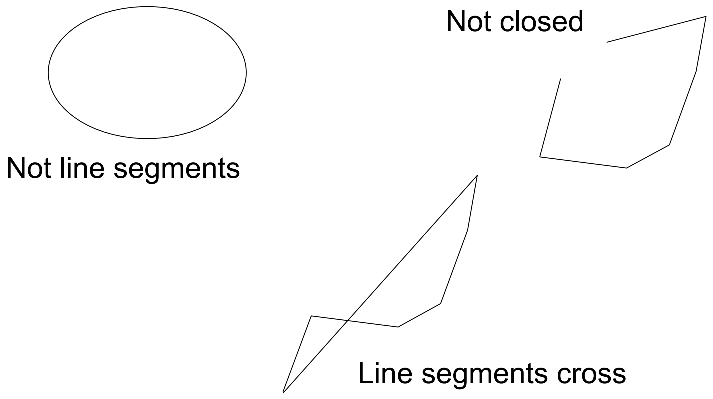
Polygons

- Polygons are 2-D objects described by three or more points connected by *line segments*.
- They are *closed* (all line segments connected) and line segments do not cross each other.
- A point used to describe a polygon is called a *vertex* (plural: *vertices*).

Example of Polygons

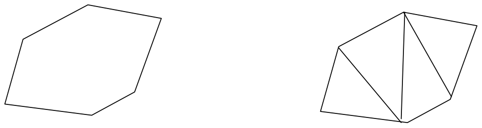


Examples of Non-Polygons



Triangles

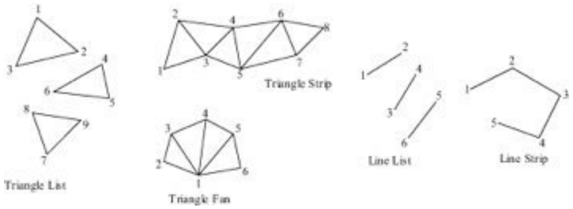
- Simplest polygon
- Nice mathematical properties
- Can divide up more complex polygons into triangles (*tessellation*)



Coordinates

- Two dimensional points can be described using 2-D *coordinates*.
- Each coordinate is two numbers (*x* and *y*), referring to horizontal and vertical position.
- For our surfaces, we must describe points in 3-D space using 3-D coordinates (*x*, *y*, *z*) referring to horizontal, vertical, and depth.
- Our polygons (triangles) will be described by 3-D coordinates.

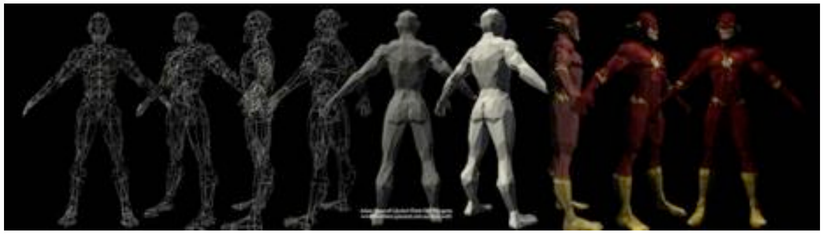
Lots of Triangles



© Charles River Media

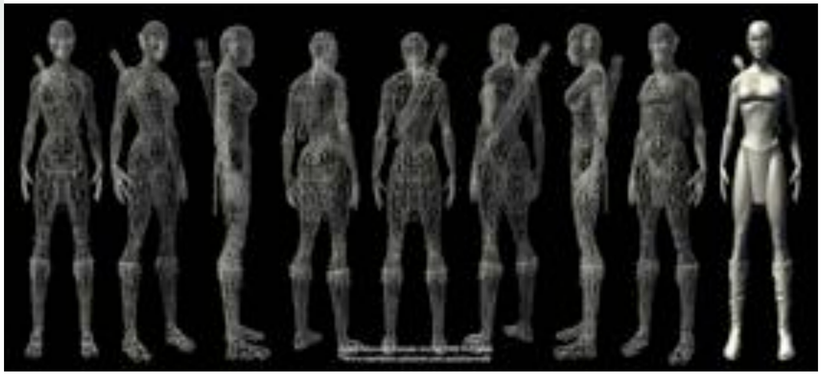
Meshes

- A collection of triangle for a single “object” is called a *mesh*.
- In the end, a mesh is just a big list of vertex coordinates

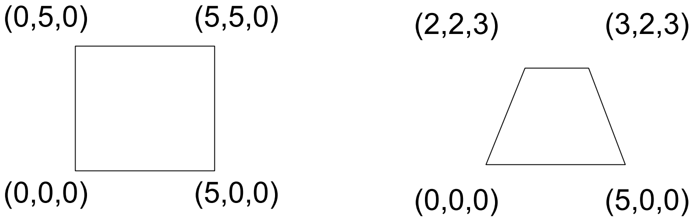


Multiple Meshes

- Figures can be composed of multiple meshes. The pieces (meshes) can separate.



2-D Object in 3-D Space



(0,5,0) (5,5,0)

(0,0,0) (5,0,0)

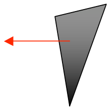
(2,2,3) (3,2,3)

(0,0,0) (5,0,0)

(These numbers are a filthy lie)

Normal Vectors

- The idea was to make “hollow” figures from surfaces built out of polygons.
- Only care about the “outside” of the figure.
- Triangles (often) have only one visible face
  - Look at it from one face, see the triangle
  - Flip it over and it’s transparent
- Conceptually, there is an arrow called the *normal vector* pointing straight out of one face to indicate that it’s the visible face.



Last time…

- Finished 2-D graphics
- Specialization and Generalization
- Started 3-D graphics
  - Types of rendering
  - Cameras
  - Polygons, Triangles, Vertices, and Coordinates
  - Meshes, Normal vectors

This time…

- How to render 3-D scenes
  - Models
  - Culling
  - Volume Partitioning
  - Advances in 3-D
  - Shading and Texturing
  - Special Effects
- Genres as time permits

How to Draw a Scene

- World is built out of many, many triangles
  - Place the camera in the world
  - Determine what triangles can seen.
  - We can calculate what a triangle looks like from the camera's position
  - From a particular viewpoint, some triangles obscure others.
- One idea:
  - sort triangles by how far away they are
  - draw them from furthest to closest
  - closer triangles will be drawn over top of further triangles

Problem

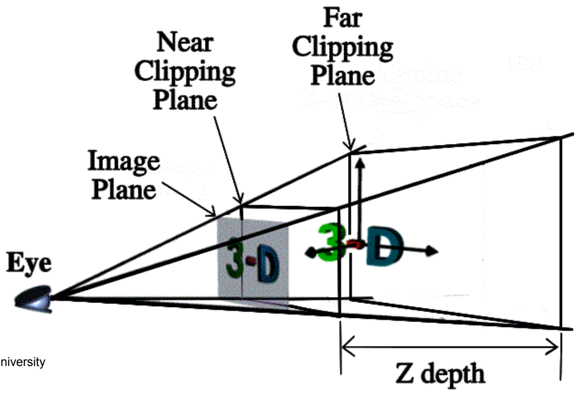
- In a large world, way too many triangles
- Limited in the number of triangles we can draw per frame
- Essentially the same problem we had in vector graphics (and 2-D raster too).
- Always need to consider *scene complexity*
  - many factors to complexity
  - number of potentially visible triangles
  - effects
- How can we have complex worlds but still draw them?

Culling

- Need to reduce the number of triangles we need to draw
- *Culling* is the process of eliminating triangles from the list of those we need to draw.
- Can obviously ignore
  - triangles behind the camera
  - triangles out of the forward field of view up-down and right-left
  - triangles beyond the camera's range
  - all this is called *frustrum culling*
  - only consider triangles inside the camera's frustrum

The Camera: Frustum

- Volume visible to camera: *frustum*



© George Otto, Penn State University

Clipping

- Only considering triangles inside the frustum now
- Triangle may only be partially inside
- Cut off part of triangle that is outside
- This process is called *clipping*

Still have problems…

- With complex worlds, even computing what triangles are inside the frustum may be too slow!
- Camera’s max depth may cover most of the world
- So, even if we can do frustum culling, we may have way too many triangles to draw
- Need a fast way of deciding what triangles might be visible before we cull
- Let’s break the world up into manageable spaces

Philosophical Question

- Ever wonder why you can’t just blow it all up?

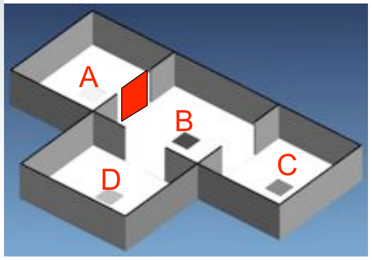
Static vs. Dynamic Objects

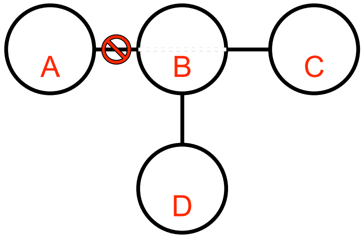
- Suppose an object in the world is *static* (doesn’t move) (e.g., a wall, a non-killing-you statue)
- We can *precompute* (calculate ahead of time) and store information about the visibility of its triangles (do this when developing the game)
- We can use the information to speed up rendering
- If an object moves (is *dynamic*), then we can’t always do this
- If most of the world is static, life will be good
- Sadly, this means we can’t change much of the world

Volume Partitioning

- One way to exploit static environments is to divide up (*partition*) the world into separate spaces (*volumes*).
- Rooms are a natural way to do this.
- We can manually specify the division when we design the levels
- There are more automatic methods, but the ideas are easiest to explain in a *portal* system.

Portals





Portals

- Divide world into regions connected by *portals*.
- Camera is limited to triangles in the same region as it.
- Can extend this by allowing view through portals into other regions.
- Extend further by allowing portals to be open or closed during the game.

Automated Volume Partitioning

- Automated methods split the world's volume up into pieces using geometry.
- Examples include
  - binary space partitioning
  - quadtrees
  - octrees
- Many games use a mixture of manual and automatic methods

Early 3-D

- Many early 3-D games were made possible by volume partitioning
- 2-D graphics (sprites) in a 3-D environment
- Ultima Underworld (Looking Glass, 1992)
- Wolfenstein 3D (id Software, 1992)
- Doom (id Software, 1993)
- Heretic (Raven Software, 1994)
- Marathon (Bungie, 1994)
- System Shock (Looking Glass, 1994)
- Duke Nukem 3D (3D Realms, 1996)

Early 3-D: 2.5D or "Raycasting"

- Early games allowed only one "floor", at most allowing variations in the height of that floor. No rooms above rooms (except by trickery).
- Ultima Underworld allowed complex environment (angled walls, inclined surfaces) but was slow
- Wolfenstein 3D made some simplifying assumptions (e.g. all-right angles, all floors the same level) and was fast
- Doom relaxed restrictions (angled walls (right angled to floor though), variable height floors (no slopes)) while keeping the speed
- Heretic allowed angling viewpoint up and down

Beyond 2.5…

- System Shock (Looking Glass, 1994) broke 2.5-D and allowed floors above other floors and transparency (a bit slow) (this functionality might have been around in Ultima Underworld… 1992!)
- But, the id games were ultimately more successful
  - simpler
  - faster

Models

- Not everything is background…
- And sprites look weird in a 3-D world…
- So we need some dynamic 3-D objects
- These are sometimes called *models* to distinguish them from static background
- Used for
  - game creatures, power-ups, doors
  - destructible parts of environment

Models

- Made of meshes
- Can be animated by changing the vertex coordinates in the mesh
- Limited numbers of (visible) models because we can't precompute
- Need to keep track of where model is in volume partitioning
- This is why back in sprite-based 3-D games, there were many, many enemies. In polygon enemy games, we can't do so many.

Later 3-D

- Games eventually introduced 3-D models in place of sprites
- Descent (Parallax, 1995)
- Quake (id Software, 1996)
- Quake II (id Software, 1997)
- GoldenEye 007 (Rare, 1997)
- Half-Life (Valve, 1998)
- Thief (Looking Glass, 1998)
- Unreal (Epic, 1998)
- System Shock 2 (Irrational Games/Looking Glass, 1999)

3-D Graphics Hardware

- Early 3-D games used *software rendering* (up to around Quake II, 1997)
  - Computed entire process on the general-purpose CPU – sprite hardware only semi-useful.
- 3-D *hardware rendering* was very expensive stuff developed mainly by Silicon Graphics (SGI) throughout the late 80's and 90's.
- Commercially viable for home gaming starting with the Voodoo card from 3dfx in 1996
- Now main processor feeds graphics information to the *graphics processing unit* (GPU)

3-D Hardware

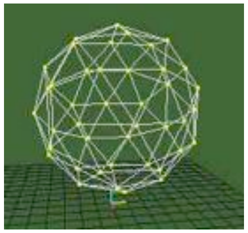
- Much of 3-D is repetitive calculations
- Custom hardware can do this more efficiently than the general-purpose CPU
- Initially, vertex information computed by CPU and sent to GPU to draw with textures
- More and more features added to GPUs
  - hardware transform and lighting
  - complex texturing operations
  - most recently, completely programmable "pixel shaders" – more like a traditional CPU!
  - expect to see physics and collision detection soon!

Rendering Triangles

- Now we know what triangles to draw
- How do we draw them?
- A little math figures out what the triangle's shape is like from the camera's perspective (*transform*)
- What do we draw?

Rendering: Wireframe

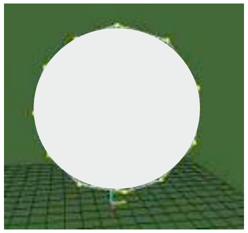
- Just draw the lines: *wireframe*



© www.realityfactory.ca

Rendering: Filled

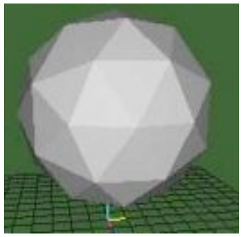
- Fill triangles with a single, solid colour
  - *filled*
  - specify colour for each triangle with RGB values
  - Problem: adjacent faces of same colour indistinguishable



Need to introduce the concept of *lighting*

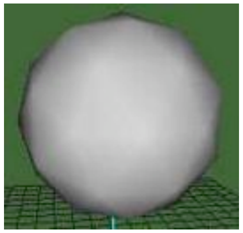
Rendering: Flat-Shading

- Draw filled triangles shaded according to light
  - *flat-shading*
  - specify colours for triangles
  - specify light(s)
  - compute effect of light on filled triangles
  - single (but different) colour for each triangle
  - gives a proper 3-D impression, but crude



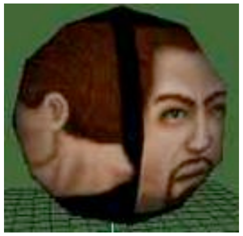
Rendering: Smooth-Shading

- Shade each triangle according to lights
  - *smooth-shading*
  - colour varies over triangle
  - smoother transitions between adjacent triangles



Rendering: Texturing

- Fill triangles with an image
  - *texturing*
  - specify entire image for surface (i.e. a colour for every pixel)
  - can shade textured triangles according to light as well
  - often call pixels in a texture image *texels*
- *texture mapping*: stretch image over an entire mesh, instead of just a single triangle



Texturing Issues

- Texturing adds a huge amount of realism
- Can add surface detail without adding triangles to the mesh
- Textures can even be animated (change image over time)
- However, requires **much** more storage than shaded triangles
- A huge part of modern gaming technology is concerned with allowing more textures, with higher resolution, and moving texture information around efficiently

Texture Resolution

- An image has a resolution
- If you get very close to a texture, it may look "blocky" because you can see individual texels
- Far away objects don't need much detail
- Standard solution: *mipmapping*
  - copies of same texture at multiple resolutions
  - switch copies as camera moves further/closer
- Switching resolutions can cause odd problems
  - partially solved by *filtering*: bilinear, trilinear, anisotropic

Level of Detail

- mipmapping is an example of a *level of detail* (or *distance*) (LOD) approach
- basic idea is that **far away things need less detail**
- we can save computation by changing what we draw when depending on distances
- other LOD approaches, include
  - changing meshes at different distances (fewer triangles when far away) (e.g. "Messiah" by Shiny, 2000)
  - dropping lights, shadows, and any other special features when stuff is far away

Transparency

- Nice to have transparent/translucent surfaces
- Can colour the light that passes through them
- Do this by adding an extra value to every pixel (texel)
  - red, green, blue, and *alpha*
  - alpha specifies how transparent the pixel is
- When drawing the scene, transparent pixels modify the value of stuff behind instead of overwriting it
- Keep in mind that transparency means more visible triangles!

Materials

- Seem to be collecting a lot of information to describe triangles
- Information about drawing triangles is often called *material* information
  - colour/texture
  - transparency (alpha)
  - reflectivity
  - special effects (e.g. bump mapping info)
  - precomputed effects (e.g. light maps, shadow maps)
- Games are using more and more detailed material information – it all costs storage!

Lights

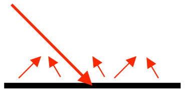
- A very complicated subject
- Basic properties
  - intensity (brightness)
  - colour
- A few kinds of lights
- ambient light
  - artificial concept of light that is “everywhere” in the scene – all surfaces are lit by it equally
  - useful so that everything is visible to some minimum extent

Lights

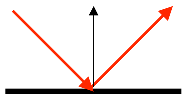
- point-source lights
  - placed in the world (like the camera)
  - position
  - shine in all directions
- directional lights
  - placed in the world (like the camera)
  - position and direction
  - shine in a cone
- dynamic vs. static
  - static lights stay where they are placed
  - dynamic lights move

Diffuse vs. Specular Lighting

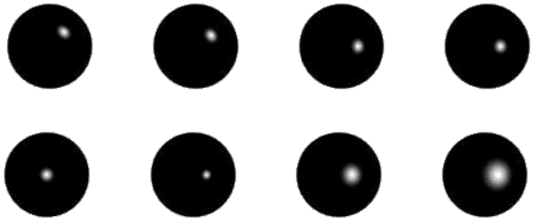
- *diffuse* lighting refers to surfaces absorbing and then emitting light uniformly over the surface
  - angle of incoming light matters (affects quantity of emitted light)
  - but viewer’s angle does not



- *specular* lighting refers to light “bouncing” off the surface
  - light comes in at an angle and leaves at an angle
  - viewer’s angle matters
  - responsible for “shiny spots” or “glare” on objects



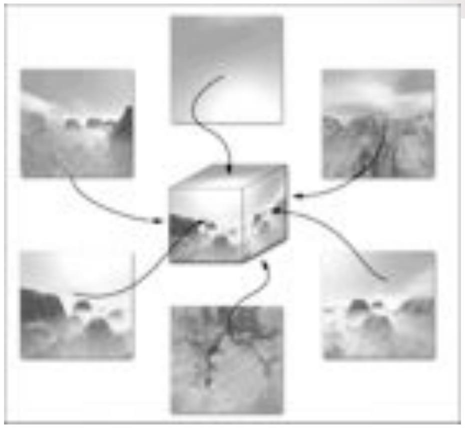
Specular Highlights



Reflectivity

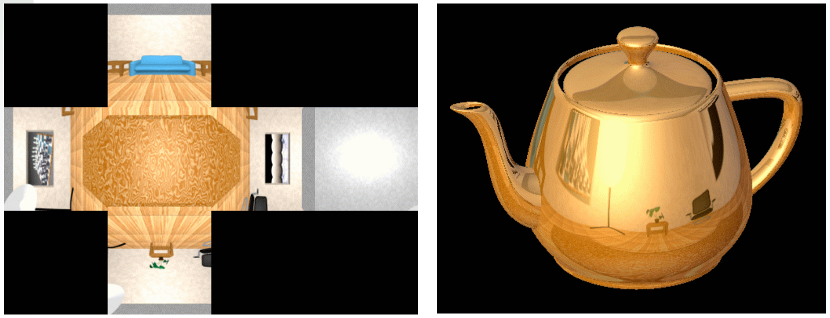
- The *reflectivity* of a material determines how much light is absorbs vs. reflects.
- Can also talk about actual *reflections*: parts of the scene reflected in shiny or mirror-like surfaces
- Expensive to do properly, so faked one way or another
  - render scene from another camera and paste the image onto the “mirror” surface (also do video cameras, etc. this way) – good but slow, have to do a whole separate render!
  - *environment maps*: precompute images for surroundings and blend with reflective surface – fast but inaccurate and no dynamic stuff

Enviroment Mapping: Cube Map



© www.developer.com

Enviroment Mapping: Cube Map



© Stephen Chenney, U. Wisconsin

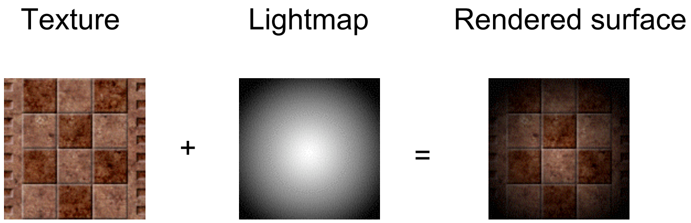
Lighting: Finer than triangles

- So far, lighting behaviour specified for each triangle
- Contemporary lighting doesn’t stop at triangles
- lighting can be *per vertex*
  - lighting behaviour specified for each vertex in the mesh and then “blended” over the triangles
- lighting can be *per pixel*
  - each pixel on the surface specifies its own lighting behaviour

Precomputing Lighting

- If lights are static, we can precompute their effect on a surface and store it
- The stored information is called a *lightmap*
- Combined with the texture of a surface during rendering to change its look

Lightmaps



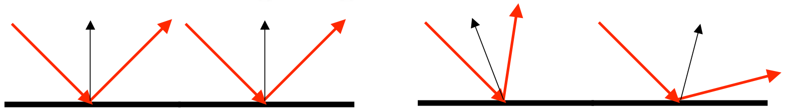
© www.flipcode.com

Lightmaps

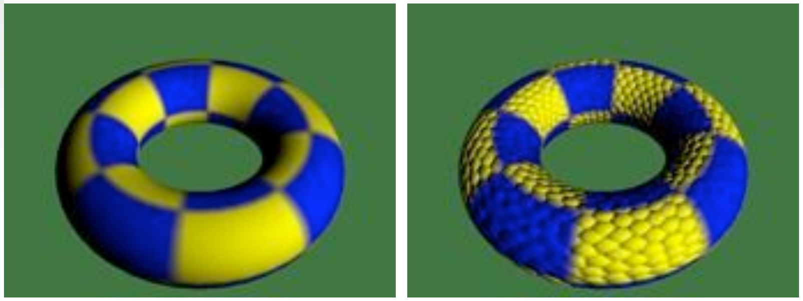
- Textures often reused and they are large
- Lightmaps are small and can be used to make textures look different in different places
- Can animate lightmaps (multiple lightmaps for a surface) – flickering lights!
- Can even generate lightmaps dynamically for dynamic lighting – slow but still good for saving storage

Bump Mapping

- A per pixel lighting effect
- Each pixel specifies its own *surface normal*
- Light is bounced off each pixel relative its individual surface normal
- Great for adding roughness to surfaces



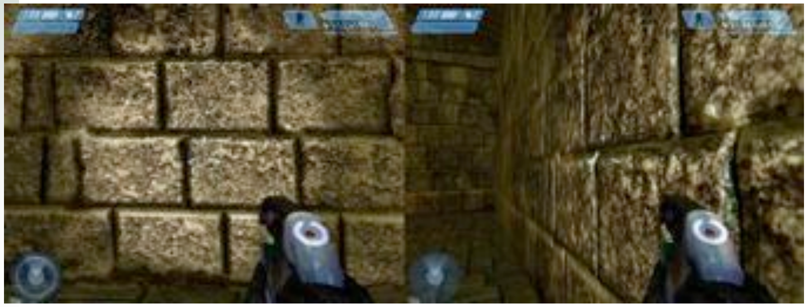
Bump Mapping



© www.earthenrecords.com

© www.earthenrecords.com

Bump Mapping: Angle Sensitive



© www.earthenrecords.com

Pixel Shaders

- General-purpose hardware in GPUs that can run a small program to decide what every pixel looks like!
- Used to create a huge range of effects.
- Much of graphics today is coming up with novel shader techiques
- Graphics hardware is increasing the flexibility of these shaders with each generation

Shadows

- Received a lot of attention lately
- Simplest: draw a dark oval under the object
- More complex shadows depend on both models and lights
- Dynamic shadows, from moving objects and lights, obviously more costly
- Shadows can be *hard* (sharp edges/uniformly dark) or *soft* (fuzzy edges)
- Very complicated subject with huge artistic value

Particle Effects and Procedural Animation

- Effects like smoke, fire, fountains are often achieved with *particles*: many, many tiny dots moved around in patterns
- Often, particle effects are a kind of *procedural animation*.
- Most animation is pre-recorded and just played back. Comes from animators and/or motion capture data.
- Procedural animation uses a program to decide how objects move. Pretty good for natural phenomena that can be described by simple laws.

Non-Photorealism

- Most graphics concerned with increasing realism
- Some efforts towards new artistic looks
- A popular example: *cel-shading*
  - named for the acetate *cels* used in hand-drawn animation
  - renders a 3-D scene to look “flatter” and more like hand-drawn
  - reduces colours and outlines in black
  - Fear Effect (Kronos, 2000), Jet Set Radio (Sega, 2000), Legend of Zelda: Wind Waker (Nintendo, 2003), XIII (Ubisoft, 2003)

3-D Collisions

- Need to decide when two models (or meshes) collide.
- Exact collisions expensive to compute.
- Approximations:
  - bounding box: find smallest box that contains each model and check whether boxes intersect
  - bounding sphere: same idea but with spheres instead of boxes
- Errors mean bits of models moving through each other or not actually touching when they stop.

Stencils

- Arbitrary 2-D cutouts
- Very efficiently implemented in hardware