Engine Programming

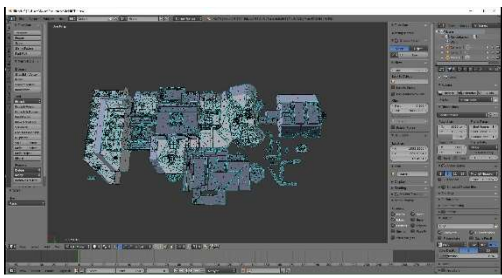
CSCI4120 Principle of Computer Game Software

Engine Programming

1. Game engine architecture
2. Rendering techniques
3. Optimization in rendering

Evolution of Game Development

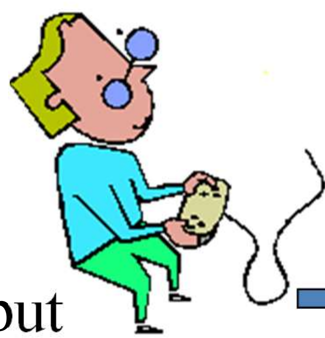
- During early 90's, the industry evolved a new approach : game engine
- Separate data from program



Level data

Game engine

display



player input

Created the "Level editing" job

What is a game engine?

- A software (library) which can provide support for drawing of graphics, sound, .. as well as other game related function supports

- Is technically difficult to develop as they involve expertise that needs mathematics, physics, and most important – programming skills

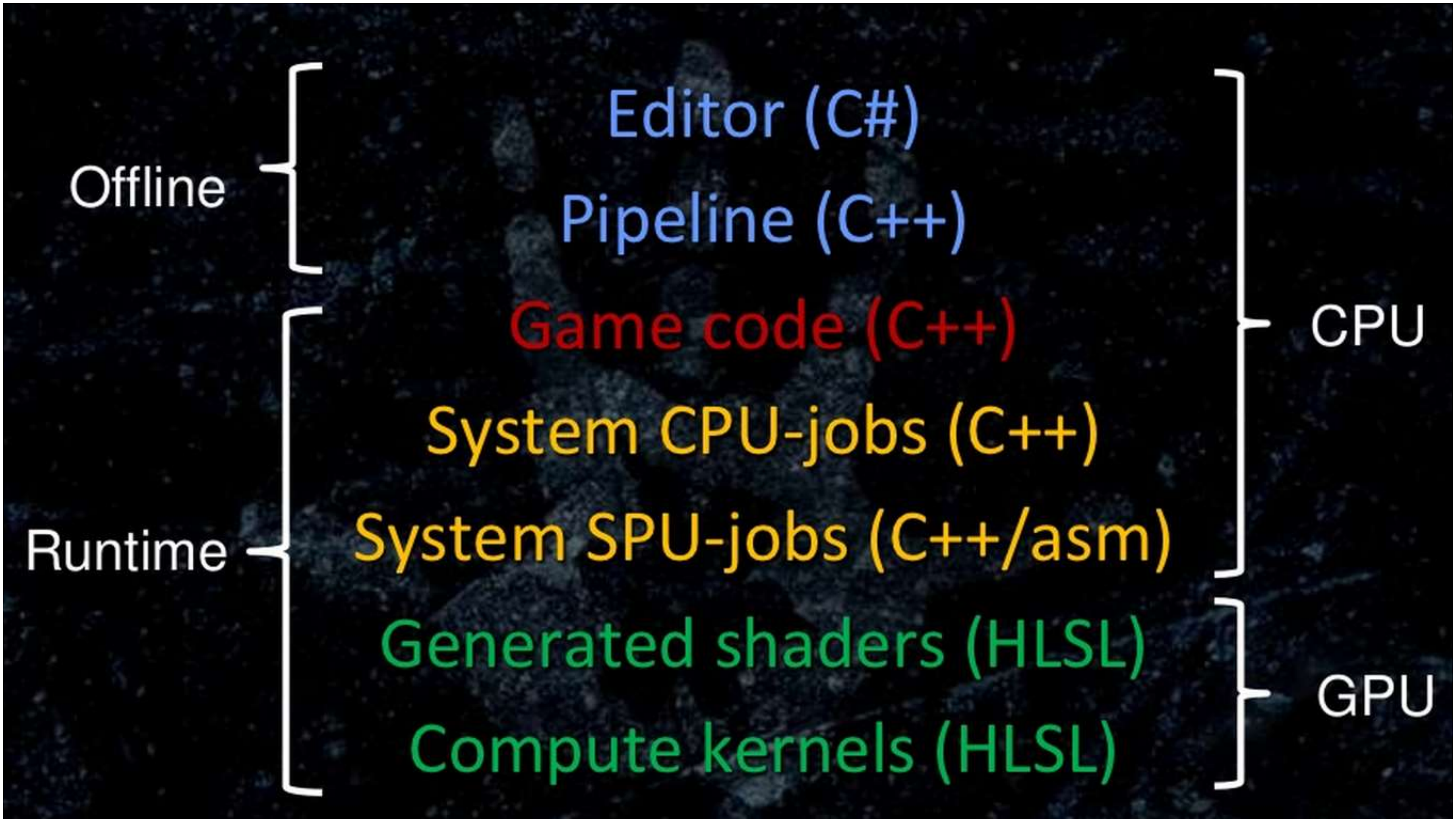
- Nowadays more focus on cross platform support e.g. PS4 & Xbox One, smartphone support

- See http://en.wikipedia.org/wiki/List_of_game_engines for list of game engines examples

What a game engine do

- Update world based on user input: *collision detection, physics*
- World rendering
- Player rendering
- Non-player characters (NPC) handling
- Network :
  message passing

Typical Code level (Frostbite)



Game Engine Approach

- Becoming the norm nowadays

- Disadvantages

1. Games produced tend to alike each other due to same inherent architecture
2. lack of creativity - game play is restricted to be the same

- Advantages

1. Better work division – artists focus on level building, programmer focus on different in-game effects
2. Player participation – enthusiast players can created customized levels
3. With different artworks, can produce another game

Middleware

- Nowaday game developers will use some specialized middleware in application development
- They usually are very good in specialized areas e.g. speedtree in tree rendering, Bink for video rendering etc.
- Many game engines just incorporated them into the engine directly e.g. Unreal

3D Game development

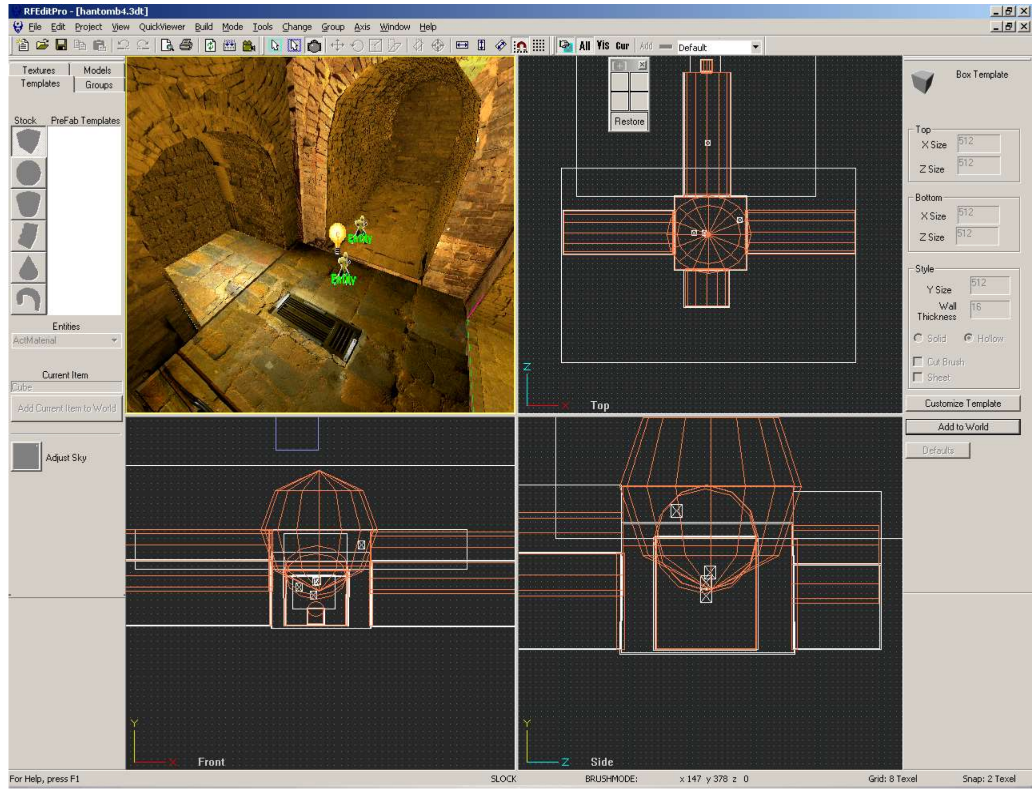
- The process is essentially similar nowadays

1. Creation/licensing of game engine
2. Use game editor to create levels (stages) of the game
3. Package with artworks & storyboards
4. Ship the product.

- Note : licensing of game engine is typically to speed up production

Game editor

- WYSIWYG game editor allows you to create a level for others to play with.
- sometimes bundled with the game/allow download to let player create their customized level eg. Snap



Editor of Reality Factory game engine

3D Game Development

- It also gave rise to MOD market

- MOD : 3rd party levels which is

1. addon levels which adhere to the theme of the game it originally based on,

Custom mod of One Punch Man in Fallout 4

3D Game Development

- A Mod can also be:

2. A total conversion(**TC**) i.e. all in-game assets e.g. characters, art, levels, are created by 3rd party developer

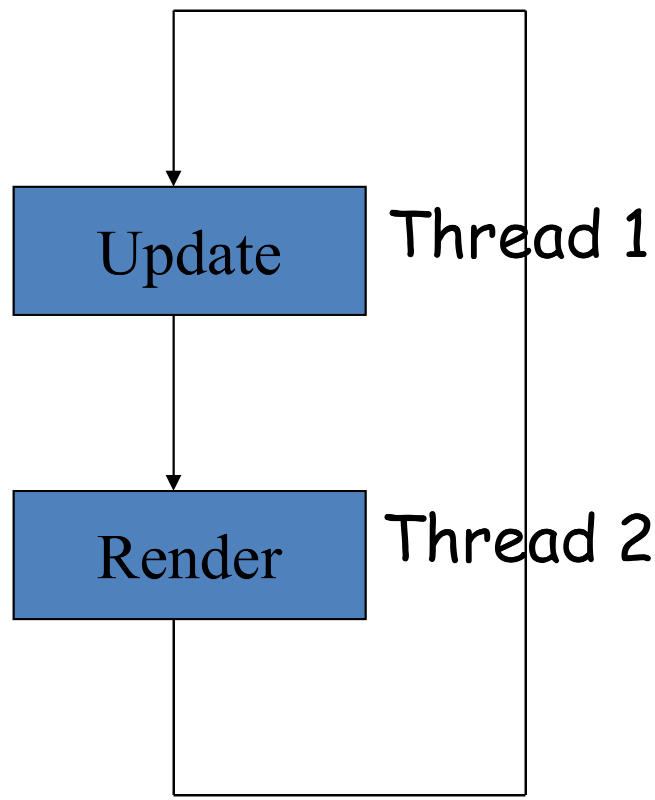
Counterstrike – a TC using Halflife

Game Engine Tech demo

- https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5

Start :A simple game architecture

- Update time is usually constant e.g. player walking speed should be independent of hardware speed
- Render rate changes according to hardware configuration
- Inherently should be implemented as multi-threaded application e.g. PC or consoles
- For single threaded machine (now rare), controlling the ratio of two calls is most common solution



Program loop

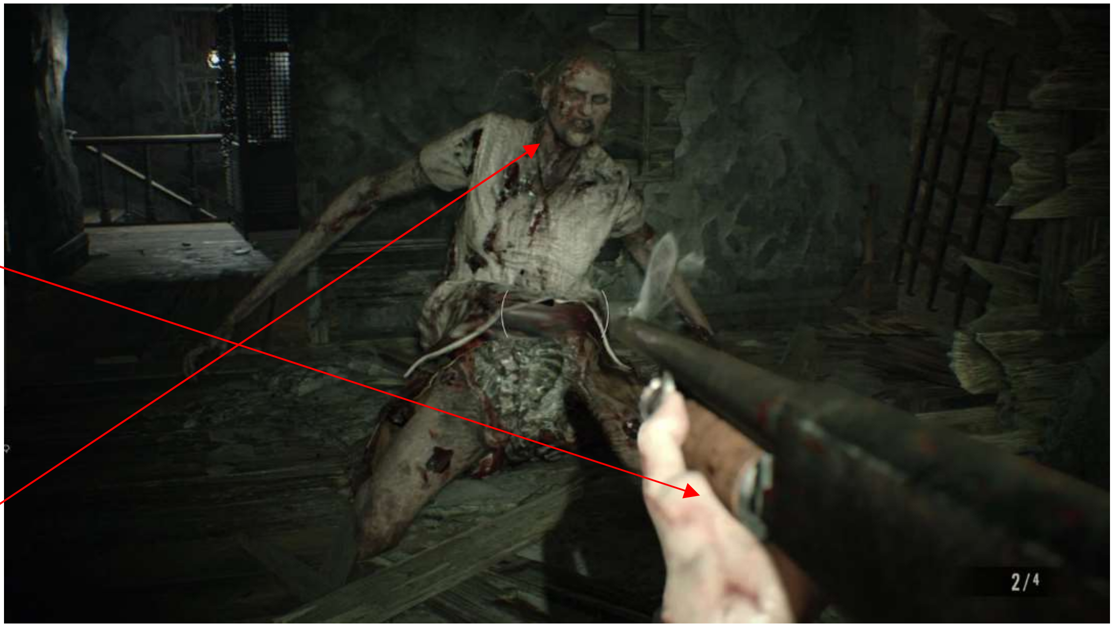
Tasks in Game Loop

- Game Logic
  - mostly the following actions:
    - player update
    - world update
    - Non-player character (NPC) update

- World update
  - consists of basically two categories of elements:
    1. *Passive*, e.g. walls, background of side scroller (physics, collision detection)
    2. *Active*, have embedded behavior. e.g. doors, enemies(AI, scripting).

Render

- World rendering – causes the most effort
- Player rendering – sometimes absent
- Non-player character(NPC) rendering



World rendering

- Usually not include game characters i.e. *level geometry* only

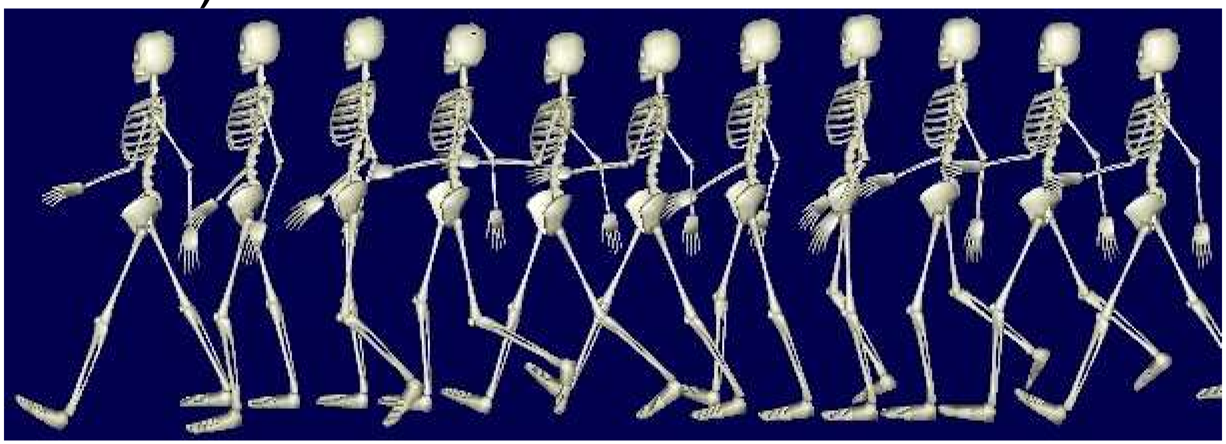
- But a typical game level is of millions of triangles

- Reduce the rendering effort through *filtering, clipping & culling (a few slides later)*

- Pruning the number of polygons to draw through
  – Visibility processing (later chapters)
  – Level of detail processing (later chapters)

NPC & player rendering

- Usually a level will have many NPCs at the same time
- Filtering also must be applied as we don't want to draw those characters not seen in current view
- Skeletal/key frame animation used (chapter of character animation)



Rendering Techniques

- Representations used
  1. Polygon
  2. Bi-cubic parametric patches
  3. Constructive Solid Geometry *CSG*
  4. Voxels
  5. Implicit surfaces

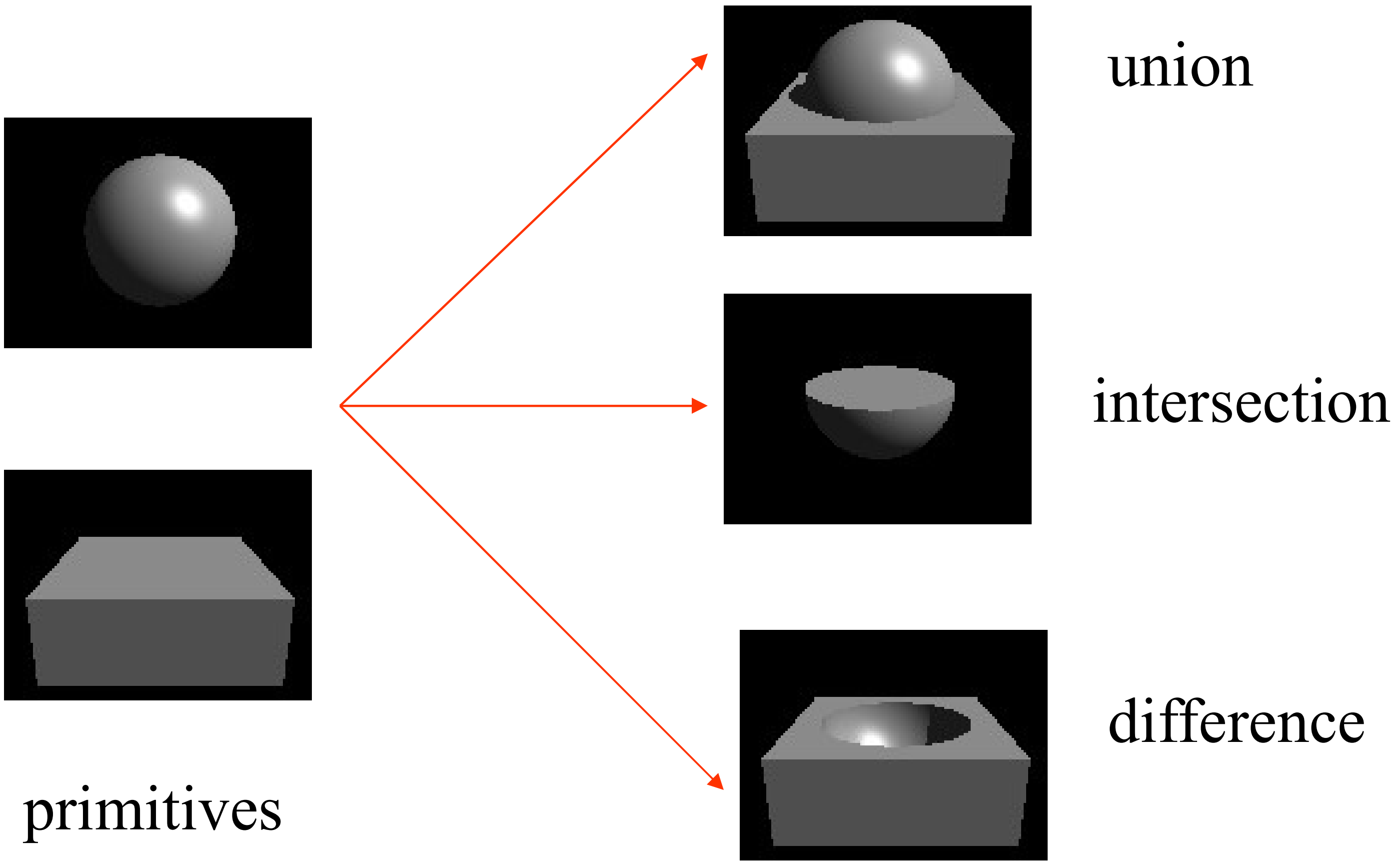
- Currently polygon representation is most efficiently rendered (hardware accelerated)

Constructive Solid Geometry (CSG)

- Consists of *Boolean* set operations on closed primitives in 3D space.
- The three CSG operations are *union*, *intersection* and *difference*.
- Produce polygon models after the modeling phase

- Used in level design in game as it is more intuitive, thus easy for artists to build complex level from basic primitives such as cone, rectangular boxes etc.

Constructive Solid Geometry (CSG)



Rendering Techniques

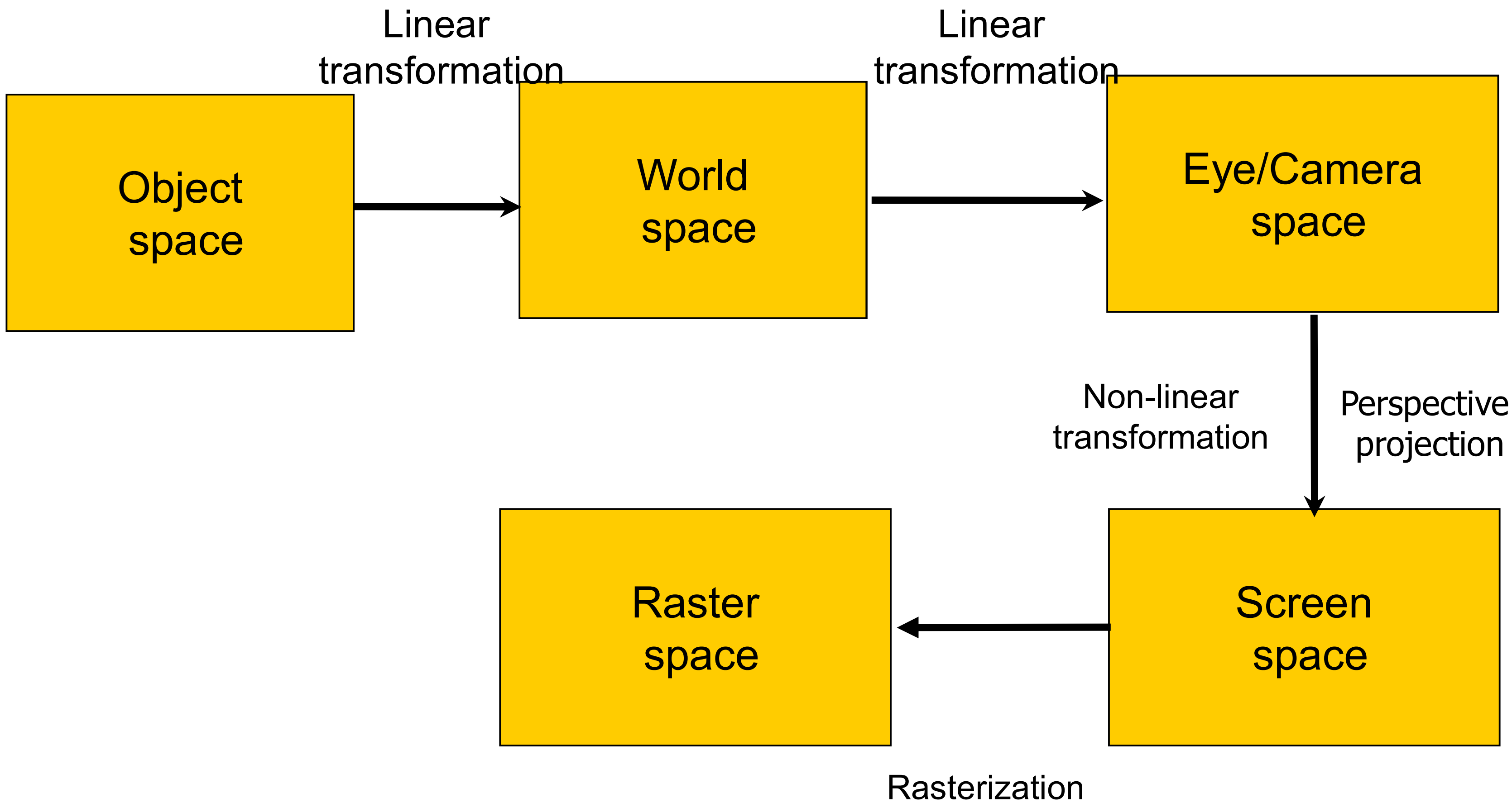
- Game engine needs to be able to render a scene with rate at least 30 frame/second in order for smooth display
- Note it also needs to take care of all updates
- Modern day 3D object triangle counts is in order of ten thousands for complex one
- We need to efficiently render

Horizon Zero Dawn's Aloy
Needs 100k tri just for in-game hair

https://polycount.com/discussion/141061/polycounts-in-next-gen-games-thread

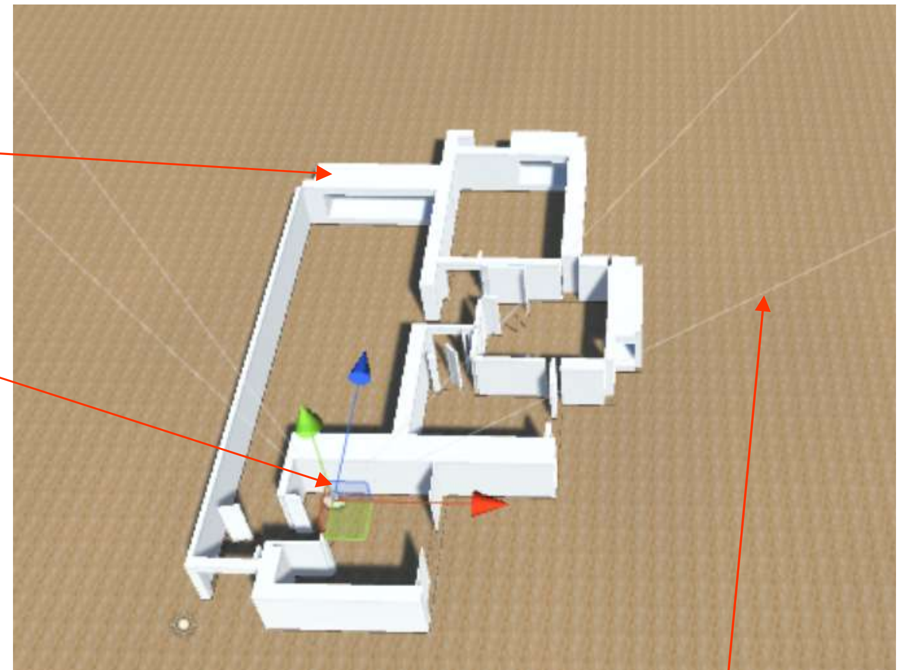
Coordinate Systems

- **Coordinate systems used when rendering geometry objects**



Rendering Techniques

- Each object (object space) is placed in the scene (world space)
- We set our player (camera) at designated position in level
- Camera space generate the resulting image
- Give a plain wall image in this case!
- Can we have the knowledge to just render the wall only to speed up?



Viewing frustum

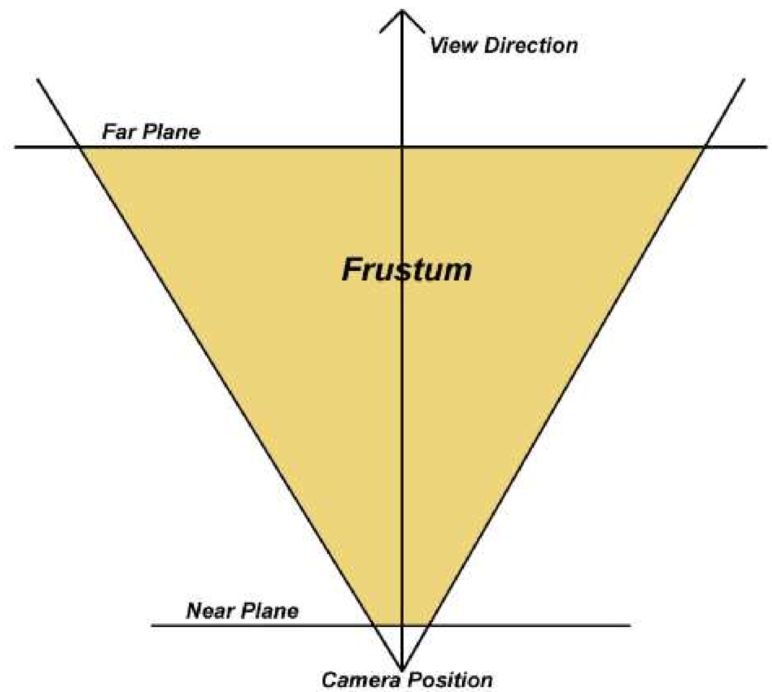
3D Graphics Pipeline

1. Visibility determination
   Clipping
   Culling
   Occlusion testing

2. Resolution determination
   LOD analysis

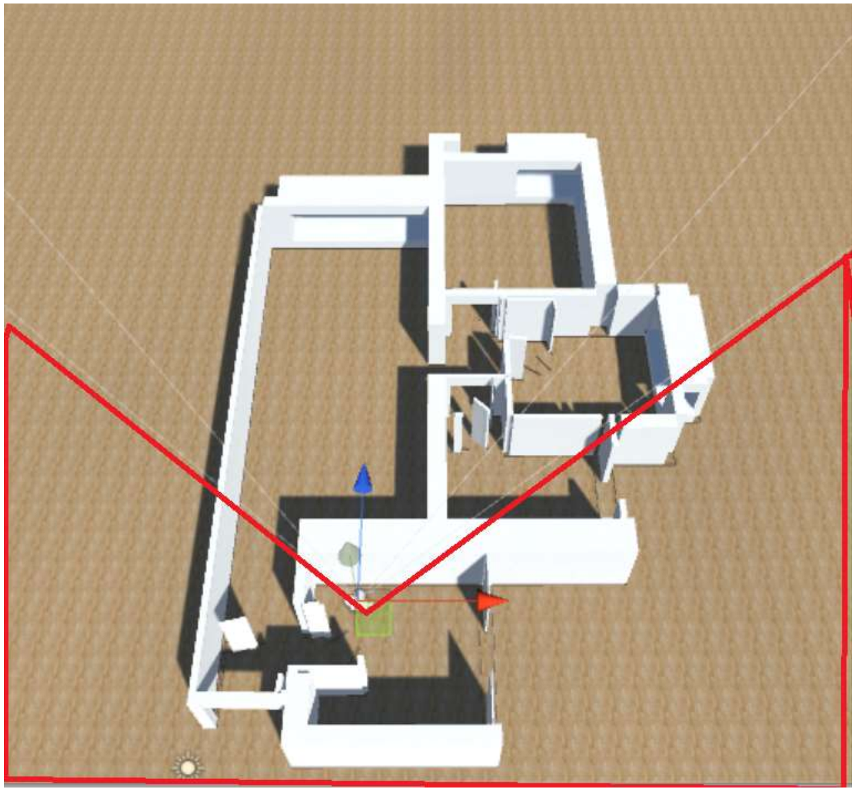
3. Transform, lighting

4. Rasterization



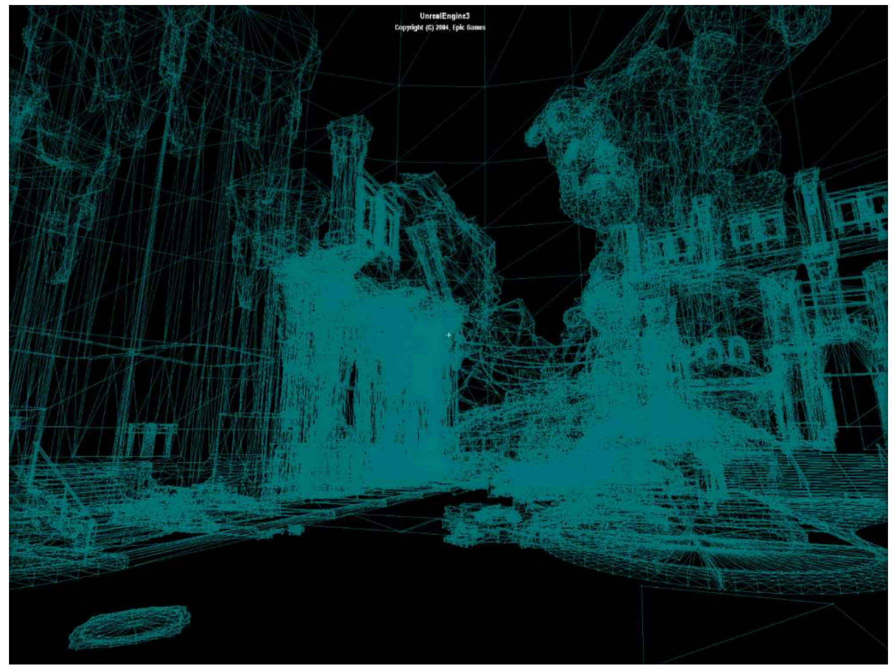
Clipping

- Eliminate unseen geometry by testing it against clipping volume(e.g. view frustum)
- Better to clip (those enclosed in red line) the geometry before passing to GPU
- Games with large level such as FPS, RPG, driving game would benefit a lot



Clipping

- All current graphics accelerators provide triangle clipping at hardware level, unseen triangles will be clipped automatically
- However sending all triangles to card would slow down rendering as it costs bus bandwidth to transmit data to card



Object Clipping

- Testing clipping at *object level* can benefit a lot as we can clip thousands of triangles at one test
- An object can be represented as a *bounding volume*
- Typically as *box* or *sphere* for easy testing
- Become the standard today as hardware performance advances

Visible Surface Determination

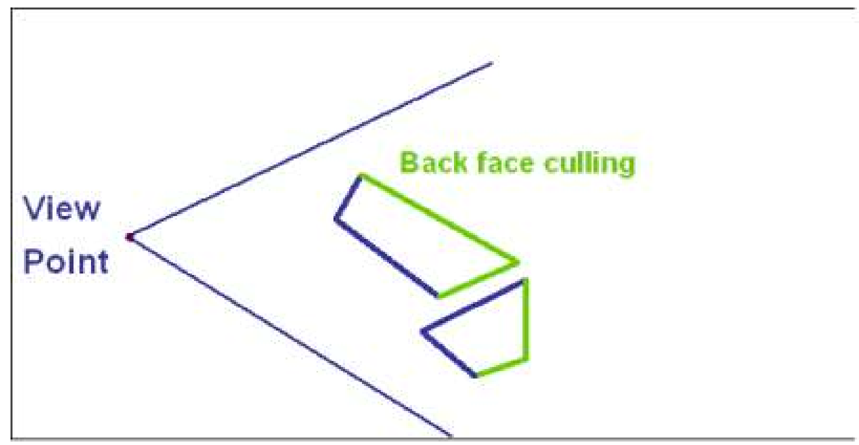
- To draw the correct picture for scene with many objects occluding one another
- We may use :

1. Backface culling
2. Depth sort
3. Z-buffer
4. Space subdivision algorithms

Backface Culling

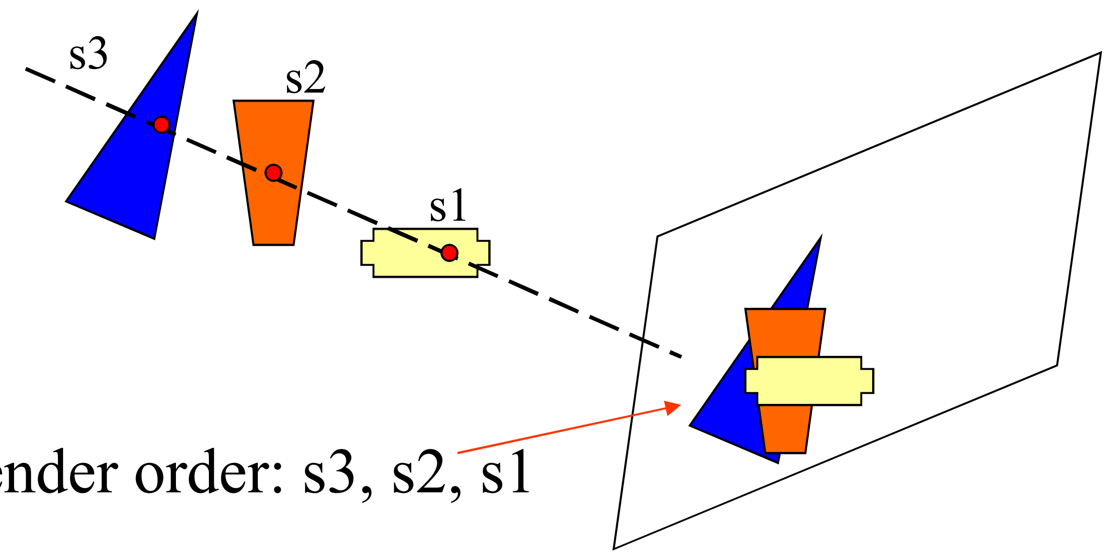
- Faces of an object with normals pointing away from camera will be occluded by other faces – can be culled (*backface culling*)

- **glCullFace(GL_BACK); // default**



Depth Sort

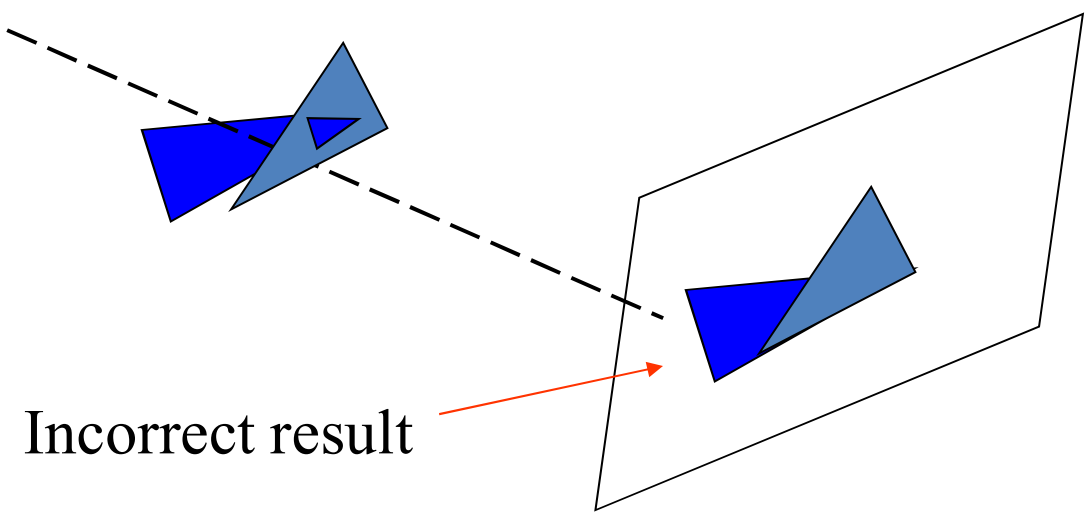
1. Surfaces are sorted in decreasing depth order using techniques such as BSP
2. Surfaces are rendered back to front



Render order: s3, s2, s1

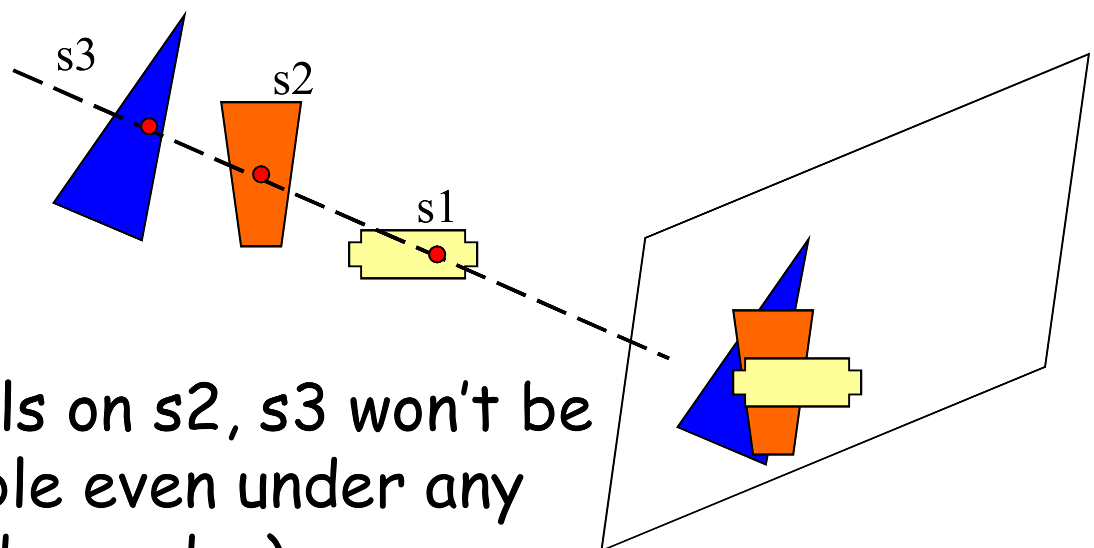
Depth Sort

- Problem : Intersecting polygons/overlapping depth give wrong result



Z-buffer

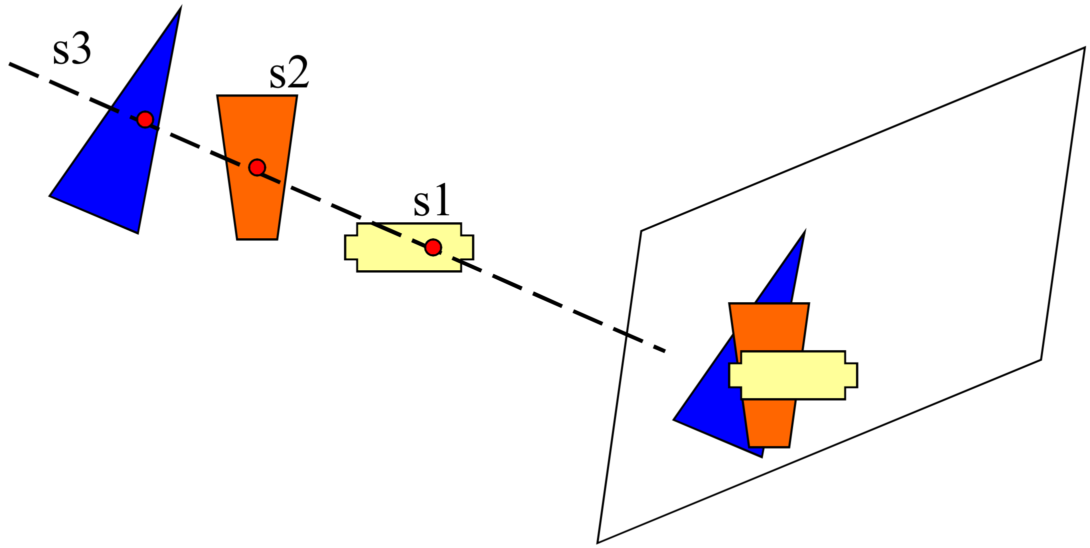
- Compare depth value for each pixel to be rendered with buffered depth of scene



Pixels on s2, s3 won't be visible even under any render order)

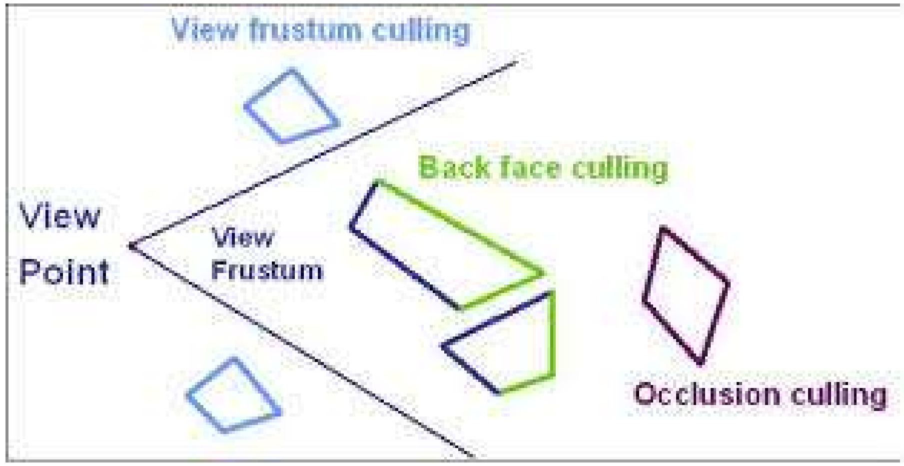
Z-buffer

- Problem: can lead to *overdraw* - consider we draw the three objects from back to front



s2, s3 are rendered to frame buffer even though a lot of pixels aren't visible in this case

Occlusion Culling



- Too much overlapping polygons would result in *heavy overdraw*
- This dramatically hinder performance in FPS game of many building
- Potentially Visible Set (**PVS**), portal rendering used in *indoor rendering* to reduce overdraw (*discussed later*)

Hardware based Occlusion Testing

- Triangle level testing is now no longer used
- Rendering algorithm
  - Draw the geometry front to back (large occluder draw first) by paint-geometry-testing:
    - If the Bounding volume(BV) not modify the Z-buffer i.e. the object fully behind other object, reject the object(skip the rendering)
- Problem:
  - overhead of each query (a draw call)
  - Latency in waiting for query result

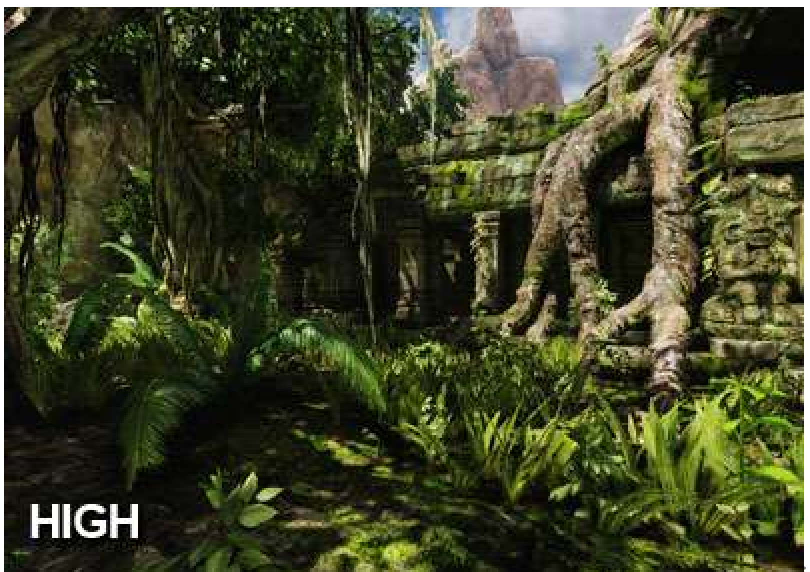
Hardware based Occlusion Testing

- Works on object level to discard large blocks of geometry in single query
- current display accelerator features:
  - Each object defined by a bounding volume
  - The BV send down the graphics pipeline to test against Z-buffer
  - A value will then indicate
    1. if the object actually modified Z-buffer
    2. If it did, how many pixels are affected

Resolution Determination

- *Resolution selection heuristic* that assign relative importance to onscreen objects – perceived *size on screen* is used typically as the heuristic

- Modern game engines now use heavily level of detail rendering to enable the program run on different hardware configurations



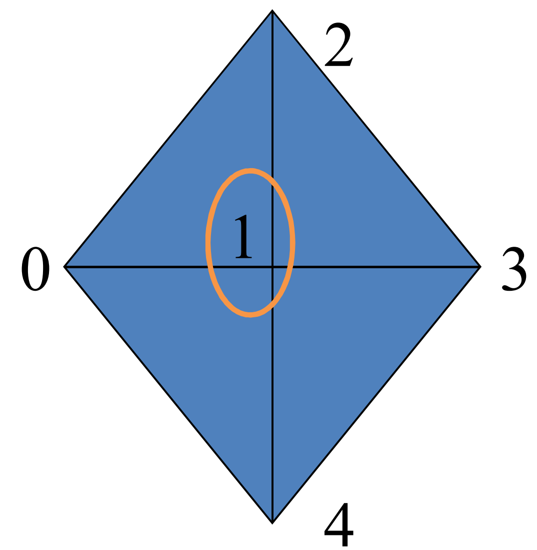
Resolution Determination

- A rendering algorithm handles the desired resolution – discrete or continuous approach to derive the rendered model e.g. **mipmapping**

- Serve to reduce workload in final rendering
- Covered in outdoor rendering

Rendering Techniques

- As a game level consists of polygon meshes with over millions of triangles, efforts must be done to optimize performance
- 3 vertices form one triangle =>
  tri 0 vert0
  tri 0 vert1
  tri 0 vert2
  tri 1 vert0
  tri 1 vert1
  tri 1 vert2
  :



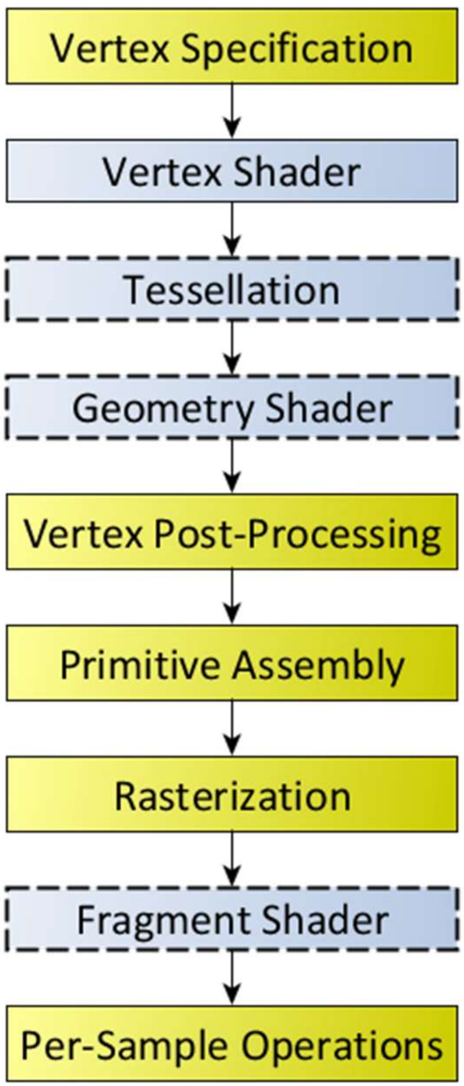
- **drawback**: has repeat occurrences for vertices shared between triangles in mesh

Modern Rendering

- *Primitives* are packed together and send as a batch – *vertex array/buffer* in OpenGL/DirectX
  - Use a *single* call to render the whole object
    Indexing primitives can further reduce the bus loading

- Write in shaders (Vertex, fragment, geometry etc.)

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview



Rendering Techniques

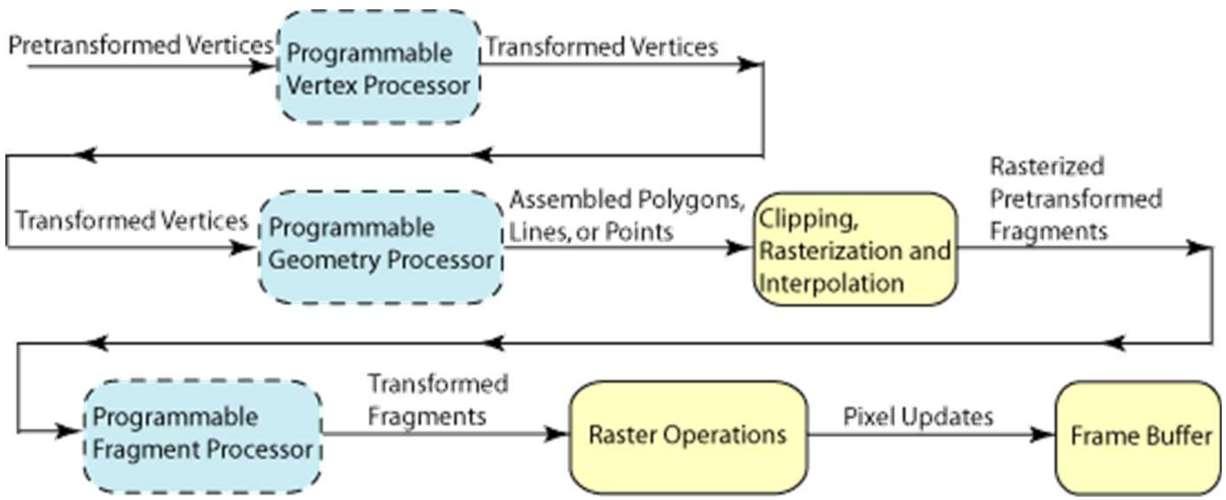
- As the hardware boost in performance, the scene details are also increasing rapidly at the same time
- Optimization in rendering still needed to produce smooth 30 frame per second animation speed
- Graphics accelerator performance limited by
  1. (Input) Bandwidth between card and CPU/main memory (sending geometry, textures etc.) – limit by interface e.g. PCI-X
  2. (Processing) GPU speed(limit by clock)
  3. (Output) Pixel fill rate

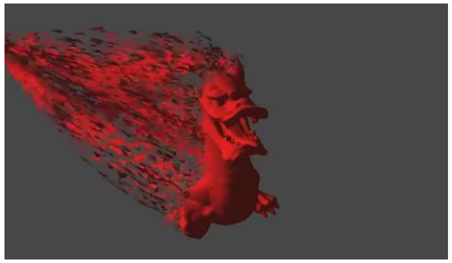
Optimization

- Bus bandwidth is the bottleneck, should try whatever to save
- *Server-side* techniques – store the geometry in the server's address space, render on demand
  - Specify multiple geometric primitives through execution of a single GL command
- **Vertex Array** – geometry send to accelerator & cached there
- **Limitation**: Not suitable for dynamic geometry i.e changing vertex position, such as animated character or procedural geometry

Shader (GPU)

- Rendering pipeline is now opened for shaders such as vertex, fragment, geometry, tessellation ..
- Shading language (GLSL) is being used
- Need to write shader code separately, compiled & binded to run time renderer.
- Can view GPU as a vector processor which process multiple data concurrently (RGBA).

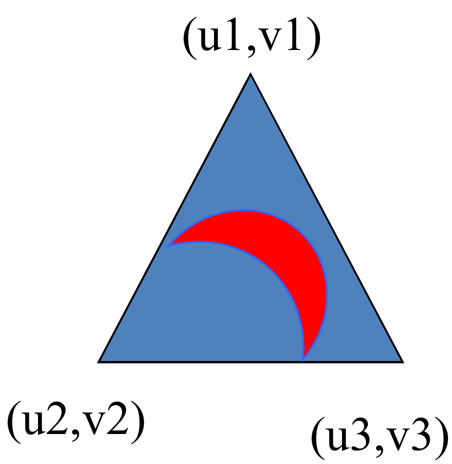


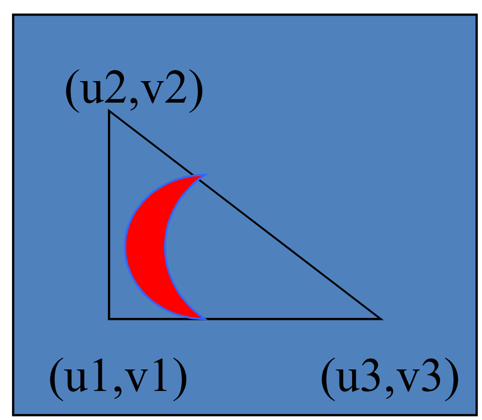


Shader demo

Texture Mapping

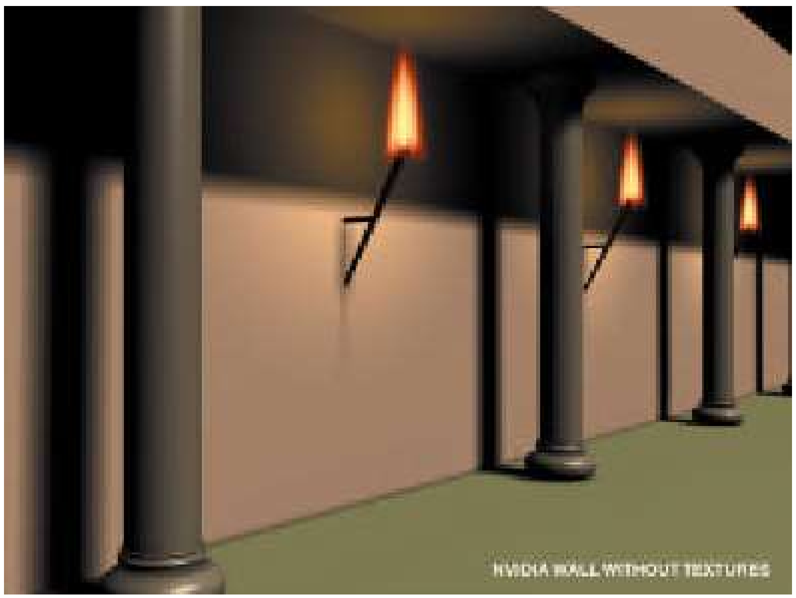
- Real surfaces consists of micro-structures with pigments, thus difficult to draw precisely
- Increase realism by mapping high details texture onto polygon surface
- Usually refers as (u,v) pair which describe position of map



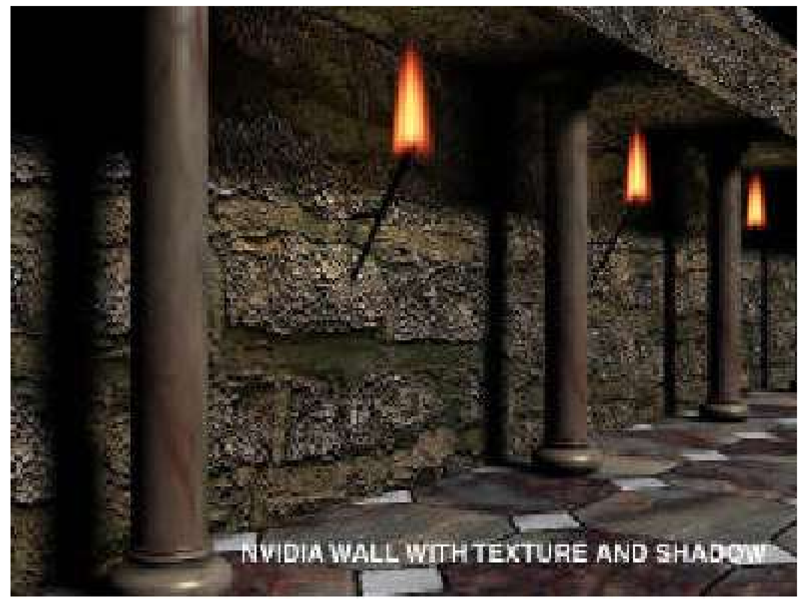


Texture Mapping

•Used extensively today to mimic surface details with little rendering costs

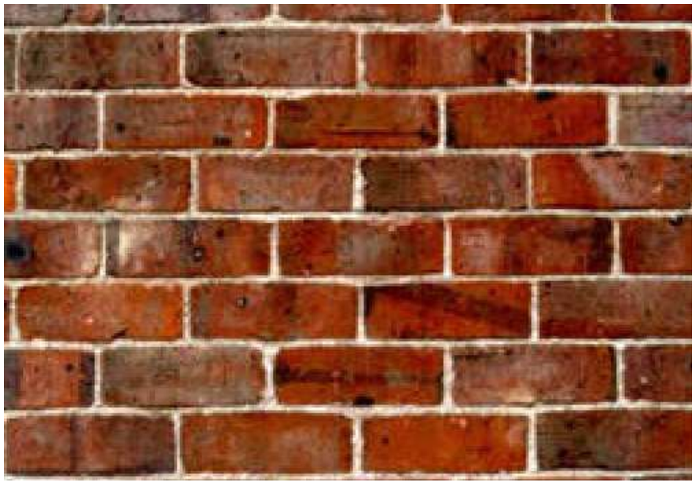


Without texture map

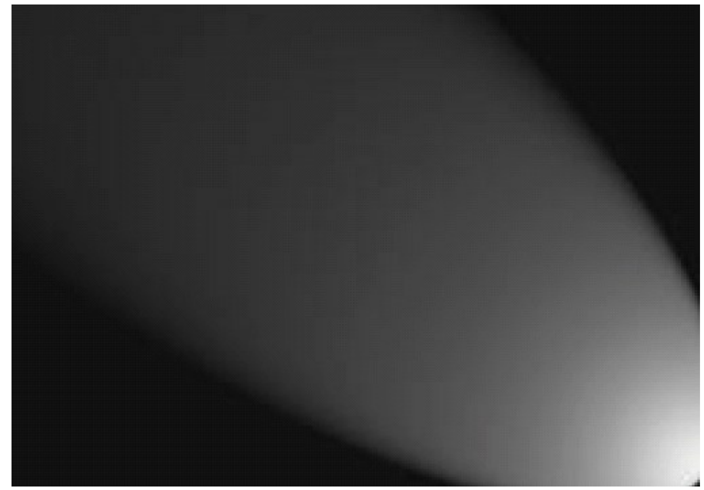


With texture map

Multitexturing



+



original

lightmap

multitexture

Introduction to Lighting Calculation

- Light sources
  - Ambient light: no identifiable source or direction
  - Point source: abstract as a point
  - Distant(directional) light : given only by direction
  - Spotlight: from source in direction
    - Cutoff angle defines a cone of light
    - With Attenuation (brighter in centre)
- Light source described by a luminance
- Each color is described separately
- $I = [I_r, I_g, I_b]^T$ (I for intensity)

Ambient Light

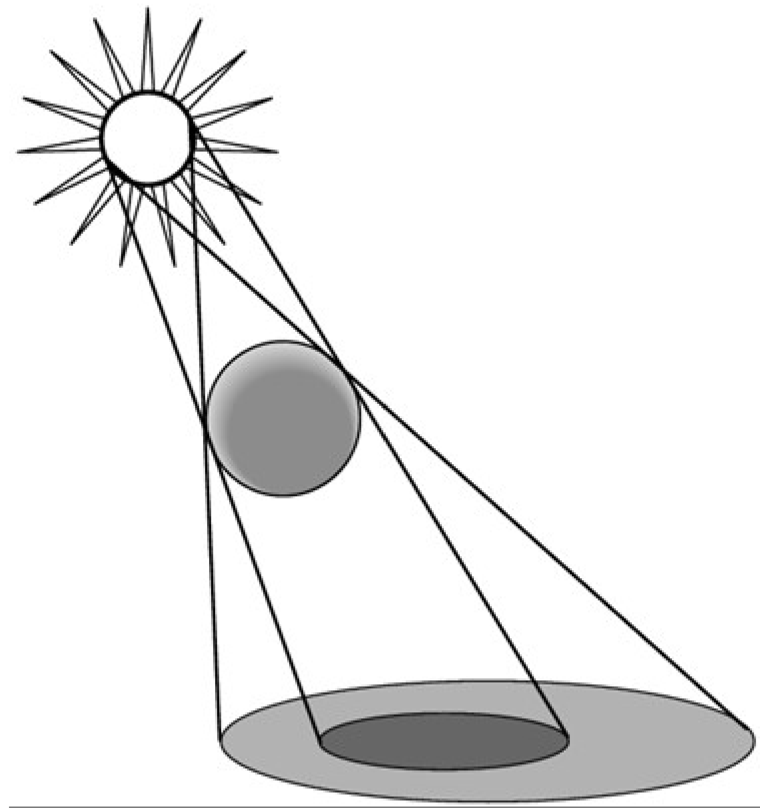
- Intensity is the same everywhere
- And the light does not have direction
- $I = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$

Point Source

- Given by a point p_0
- Light emitted from that point in all directions
- $I(p_0)=\begin{bmatrix} I_r(p_0) \\ I_g(p_0) \\ I_b(p_0) \end{bmatrix}$
- Intensity decreases with square of distance
- $I(p, p_0) = \frac{1}{|p-p_0|^2}\ \mathrm{I}(p_0)$

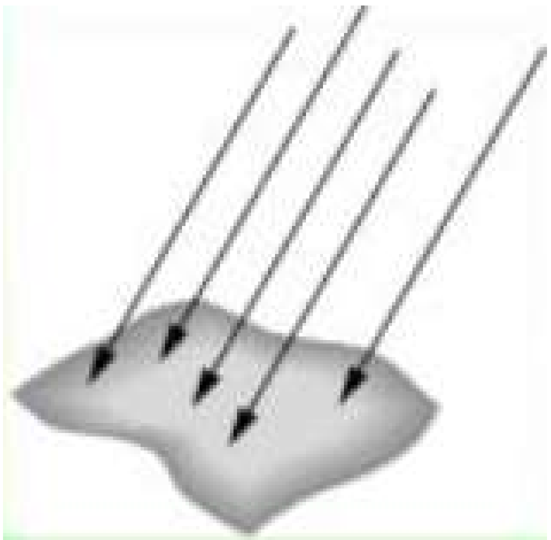
Limitation of Point Source

- Shading and shadows inaccurate
- Example: penumbra (partial "soft" shadow)



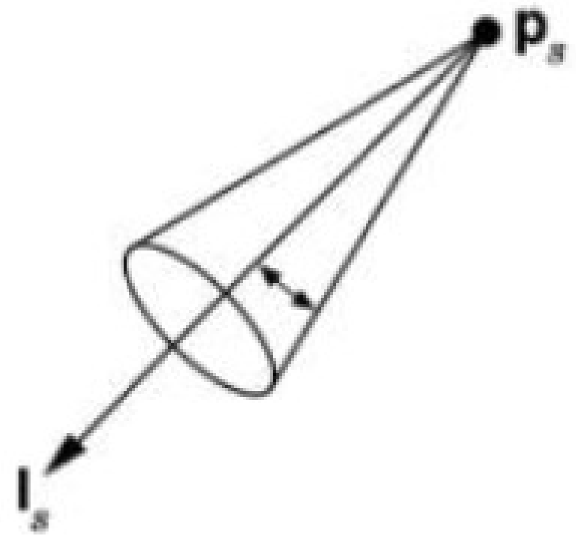
Distant Light Source

- Given only by a direction vector
- Intensity does not vary with distance (think about the sun)



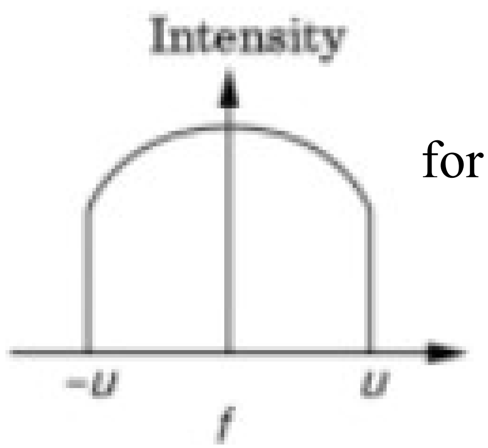
Spotlight

- Most complex light source
- Light still emanate from point
- Cutoff by cone determined by angle θ



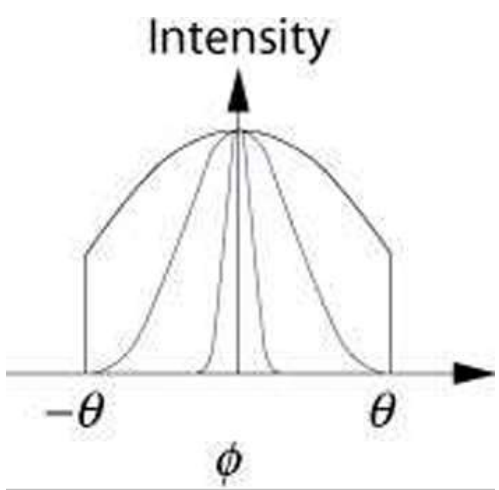
Spotlight Attenuation

- Spotlight is brightest along I_s
- vector v with angle ϕ from p to point on surface
- Intensity determined by cos ϕ
- Corresponds to projection of v onto Is
- Spotlight exponent e determines rate
- $I = cos^e$(ϕ)=(v·I_s)^e



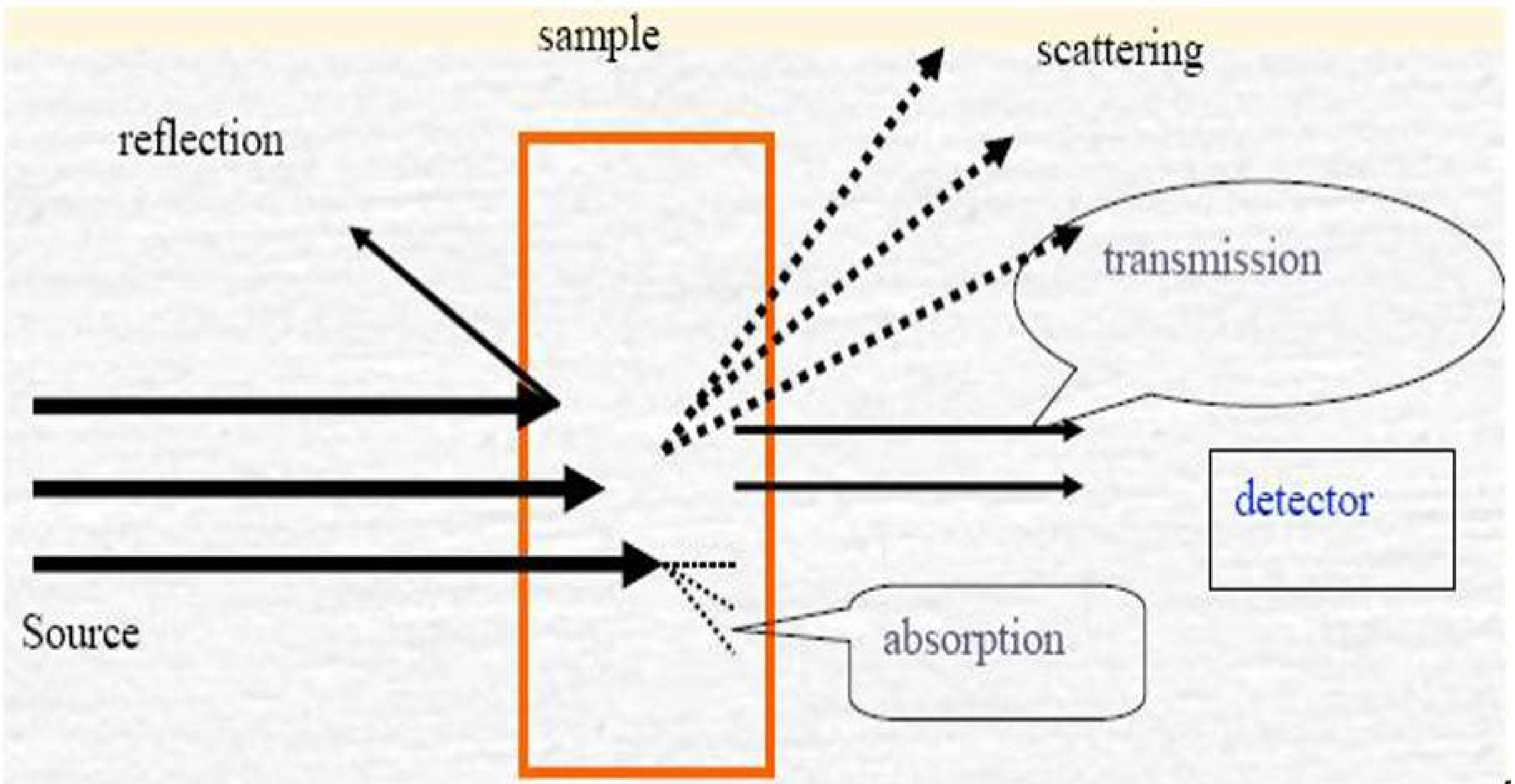
for e=1

for e>1
curve narrows



Lighting Calculation

- Now for any point in space, we can calculate illumination(lighting) arriving from various sources
- However the actual process is much more complicated



Surface Reflection

- When light hits an opaque surface, some is absorbed, the rest is reflected
- The reflected light is what we see
- Reflection is not simple and varies with material
  - The surface's micro structure defines the details of reflection
  - Variations produce anything from bright specular reflection (mirror) to dull matte finish (chalk)

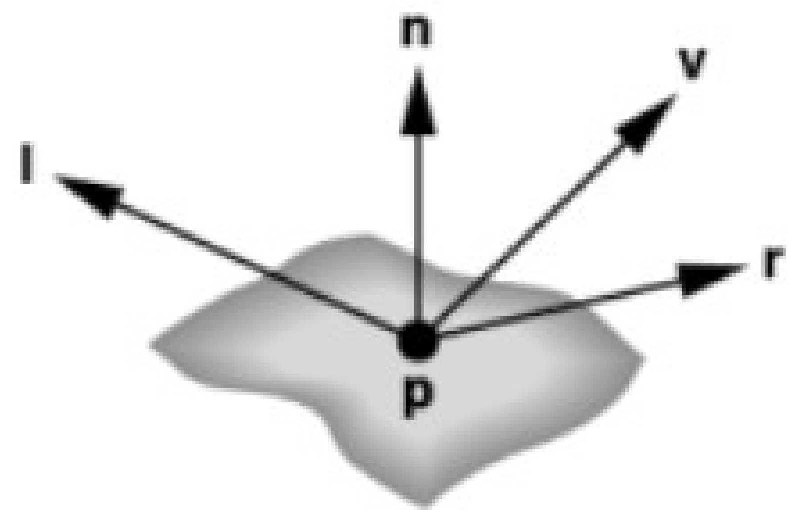
Lighting Models

- Different lighting models are considered in graphics community to model as accurately as possible the lighting in a scene
  - Phong model
  - Ray tracing
  - Bidirectional Reflectivity function(BRDF)
  - Radiosity

- Phong model most popular due to computational consideration

Phong Illumination Model

- Popular model used in computer graphics
- Calculate color for arbitrary point on surface
- Basic input are material properties and l, n, v



- l = vector to light source
- n = surface normal
- v = vector to viewer
- r = reflection of l at p determined by l and n

Basic Calculation

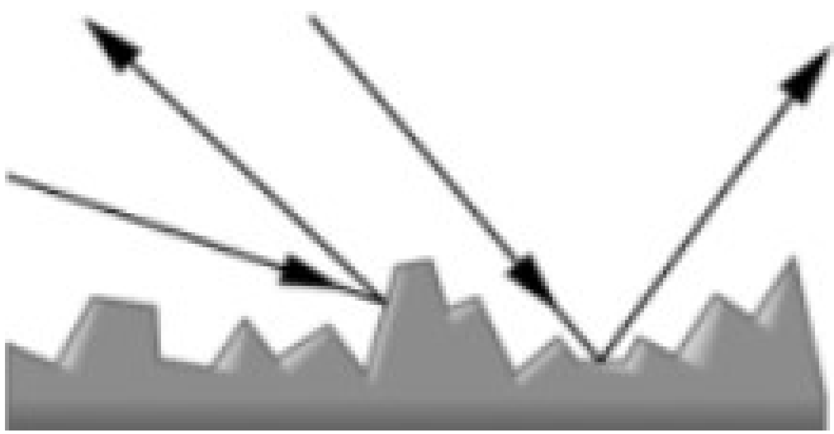
- Calculate each primary color separately
- Start with global ambient light
- Add reflections from each light source
- Clamp to [0,1]
- Reflection decomposed into
  - Ambient reflection
  - Diffuse reflection
  - Specular reflection
- Based on 3 components

Ambient Reflection

- Intensity of ambient light uniform at every point
- Ambient reflection coefficient k_a, $0 \le k_a \le 1$
- May be different for every surface and r,g,b
- Determines reflected fraction of ambient light
- Define L_a as ambient component of light source
- Ambient intensity $I_a = k_a L_a$
- L_a is not physical meaningful quantity

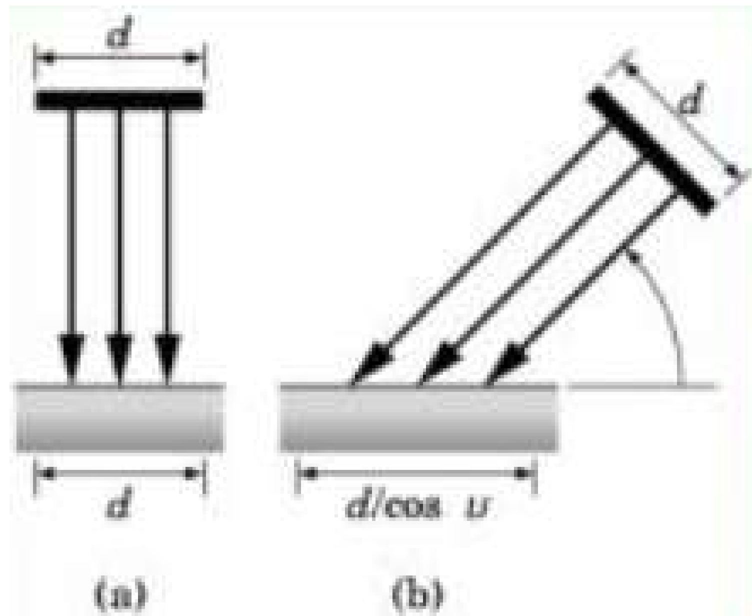
Diffuse Reflection

- Diffuse reflection models scattered light
- Assume equal in all direction
- Called *Lambertian* surface
- Diffuse reflection coefficient k_d, $0 \le k_d \le 1$
- Angle of incoming light still critical



Lambert's Law

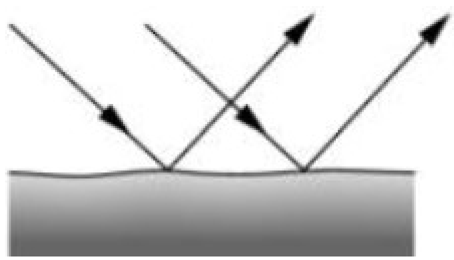
- Intensity depends on angle of incoming light
- Recall
  l=unit vector to light
  n= unit surface normal
  θ=angle to normal



- cos θ = $l \cdot n$
- $I_d = k_d (l \cdot n) L_d$
- With attenuation: $I_d = \frac{k_d}{a+bq+cq^2}(l \cdot n)L_d$
- Q: distance to light source
- L_d: diffuse component of light

Specular Reflection

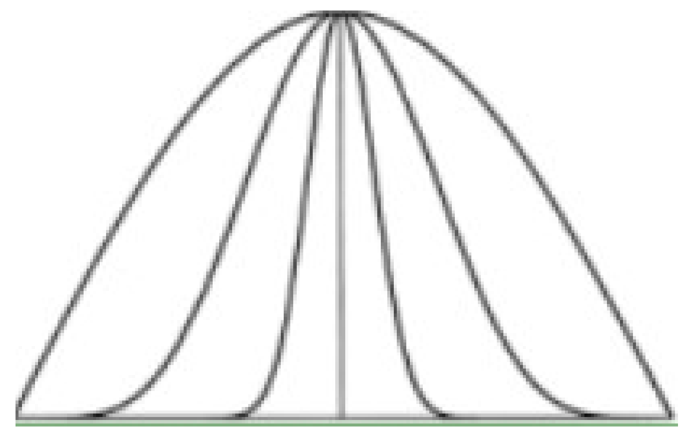
- Specular reflection coefficient k_s, $0 \le k_s \le 1$
- Shiny surfaces have high specular coefficient
- Models specular highlights
- Do not get mirror effect



Specular highlight

Shininess Coefficient

- L_s is specular component of light
- R is vector of of perfect reflection of I about n
- V is vector to viewer
- ϕ is angle between v and r
- $I_s = k_s L_s \cos^{\alpha} \phi$
- α is shininess coefficient
- Compute $\cos \phi = r \cdot v$
- r and v must be unit vector
- Also need to multiply distance term



higher α is narrower

Summary of Phong Model

- Light components for each color
  - Ambient L_a, diffuse L_d, specular L_s
- Material coefficients for each color
  - Ambient k_a, diffuse k_d, specular k_s
- Distance q for surface point from light source
- $I = \frac{1}{a+bq+cq^2}(k_d L_d(l \cdot n) + k_s L_s(r \cdot v)^{\alpha}) + k_a L_a$
- `l`: vector from light r: `l` reflected about n
- n: surface normal v: vector to viewer

Lighting in Games

- Lighting calculation in game
  - Pre-calculated static lighting - lightmaps
  - As real time lighting calculations are costly operations

Lighting

- In gaming environment, the following situations are being considered:
  - Static lights (lights do not move) – have effect on static objects, thus can be pre-calculated and stored as a texture map(lightmap)

  - Dynamic (moving) lights – the polygons being affected have to be updated with vertex or pixel lighting calculation (real time)

  - Ambient light - each polygon has an ambient light associated with it

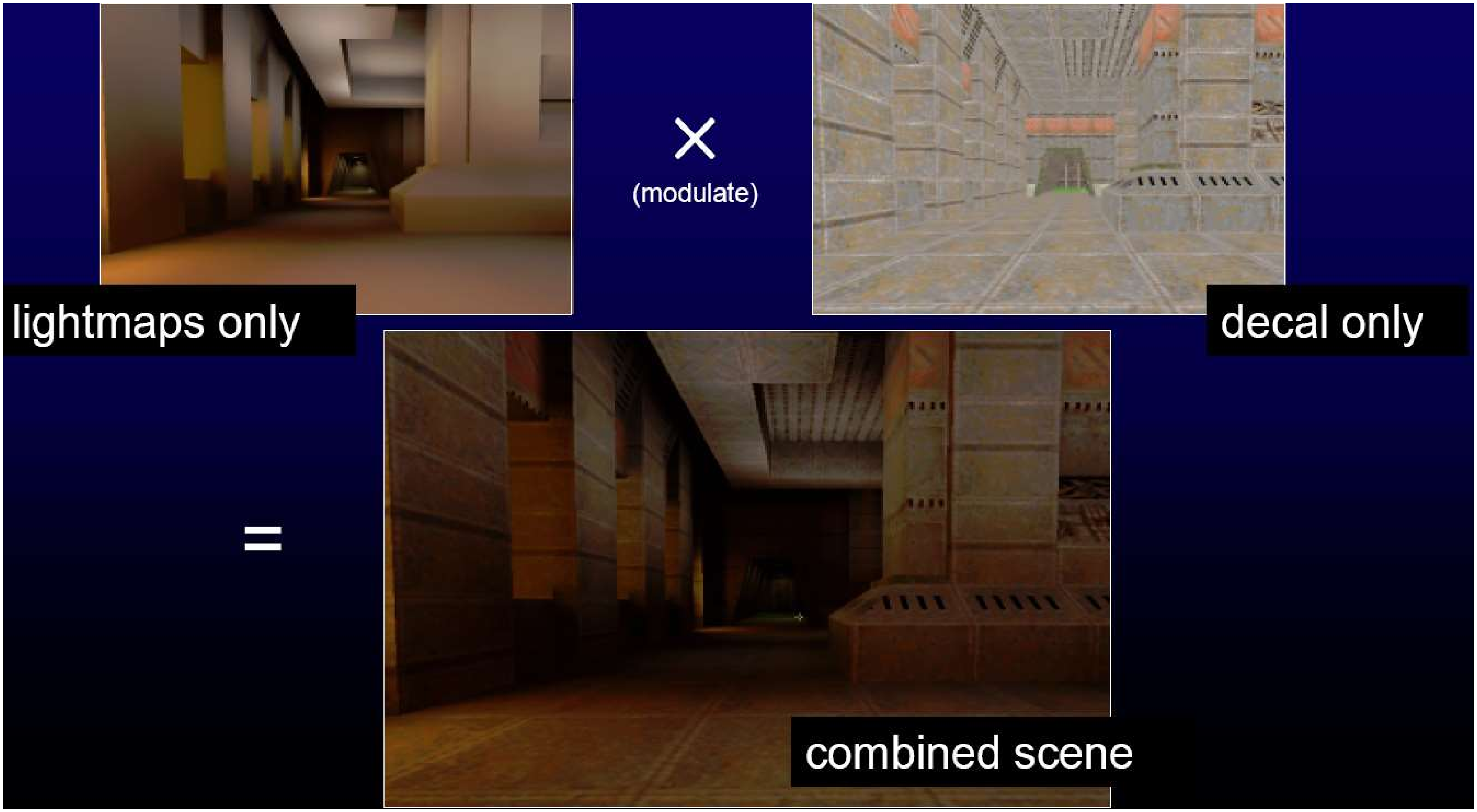
Lighting

- Real time per pixel lighting calculation for every object in scene is difficult to achieve even though on today's hardware

- **Lightmapping** : stores the lighting information in low resolution texture and *multi-textured* to form the per pixel lighting
  - Sample reflected light over a surface and store this in a 2D map
  - Allow more sophisticated **global** lighting calculation method e.g. radiosity, be used

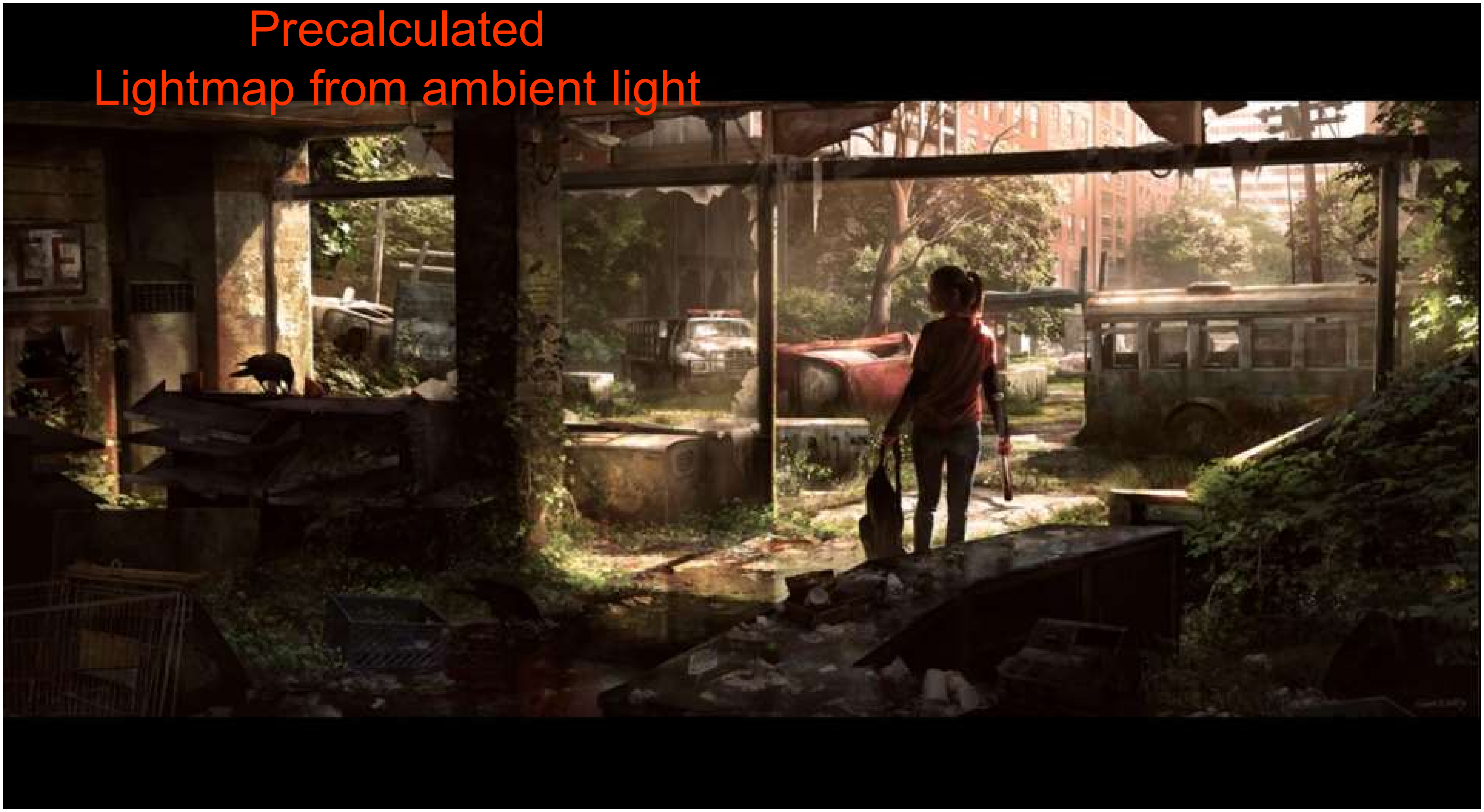
  - Little computational cost during rendering – faster than Gouraud interpolation
  - tend to be heavily magnified - permits packing lightmaps for many surface into a single texture image
  - Need long computation time for radiosity lightmap preparation

Lightmaps



Lightmapping

Precalculated
Lightmap from ambient light



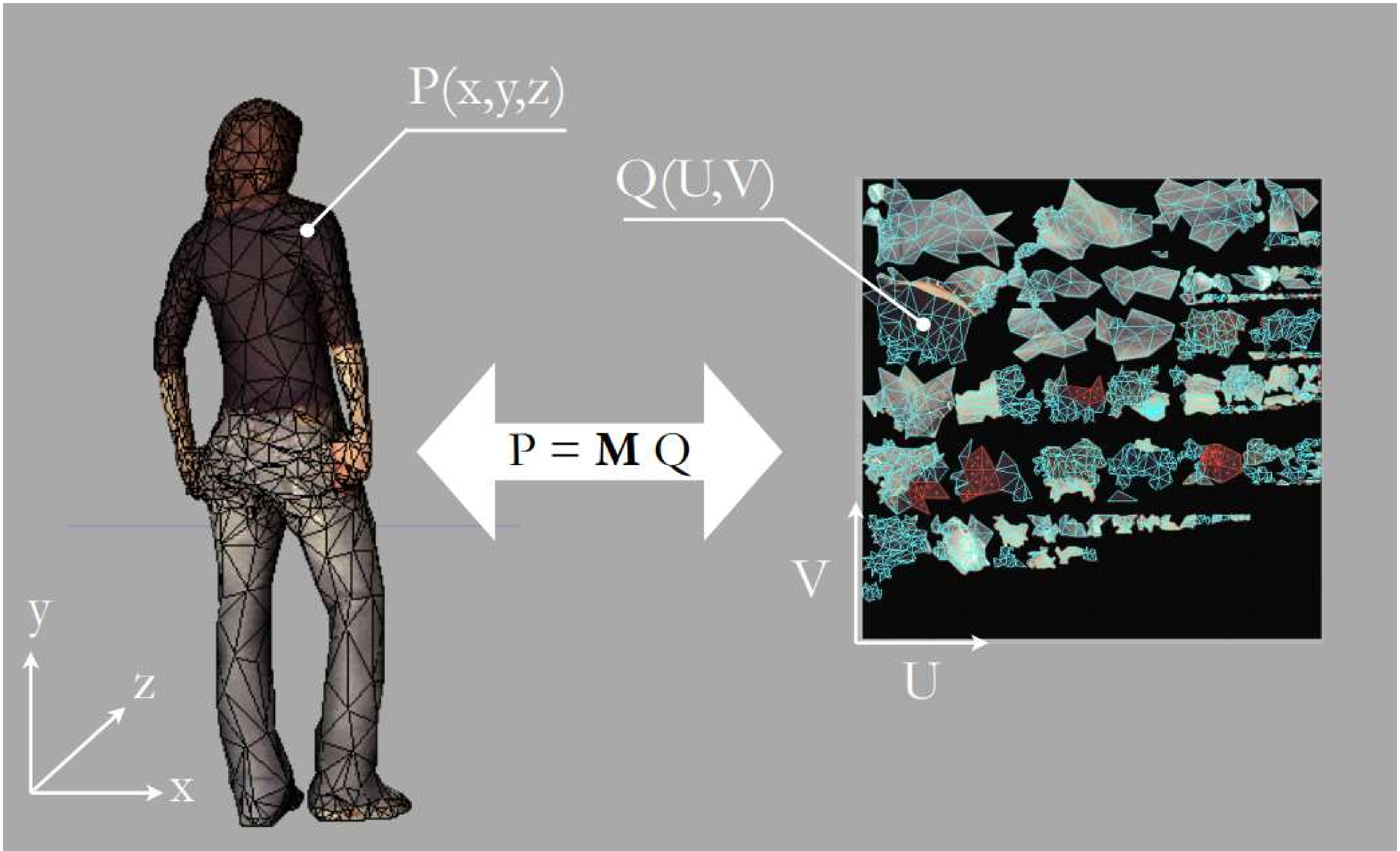
Last of Us

Lightmap

- Can either be stored separately to texture maps, or object texture map pre-modulated by the lightmap
- Typically lower resolution for separate texture as view-independent lighting changes more slowly than texture detail
- To compute lightmaps with vertex/texture coordinate association already done, we can use this correspondence to derive an affine transformation to sample light across the face of triangle
- With three non-linear points, we can find the affine matrix{a,b,c..i} from equation below:

$$\begin{bmatrix} x0 & x1 & x2 \\ y0 & y1 & y2 \\ z0 & z1 & z2 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u0 & u1 & u2 \\ v0 & v1 & v2 \\ 1 & 1 & 1 \end{bmatrix}$$

$P = MQ \quad \Rightarrow M = PQ^{-1}$

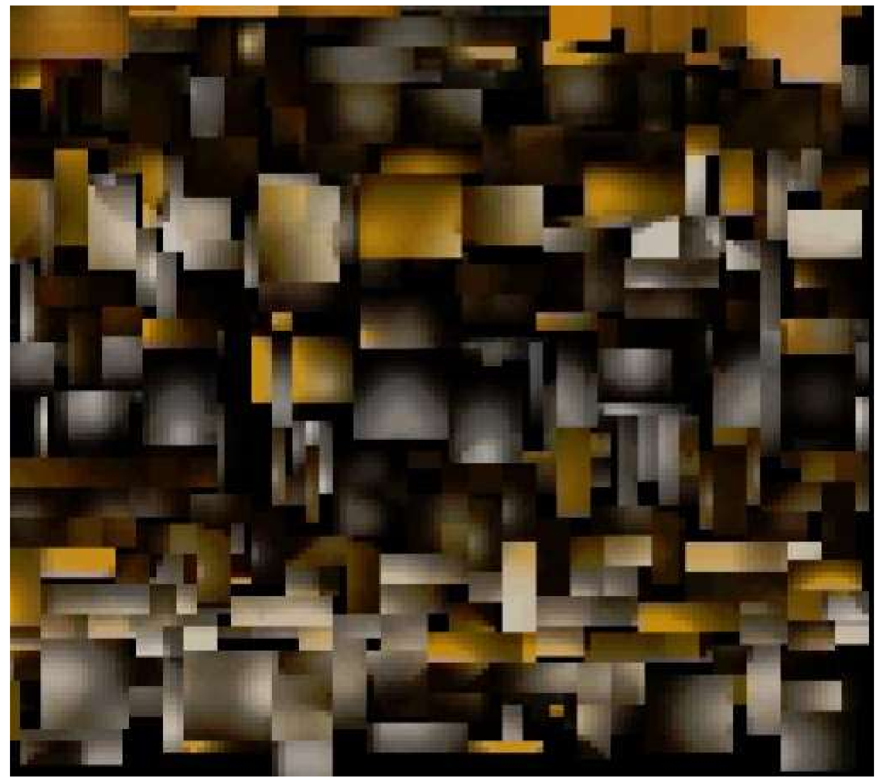

P(x,y,z)
Q(U,V)
P = **M** Q
V
U
y
z
x

Lightmap

- Using the transformation, we can scan converting the polygon projection in texture space i.e. from (u,v) to (x,y,z)

- Diffuse lighting tends to be low-frequency patterns

- Permits multiple lightmaps packed into a single texture.



Dynamic lighting with lightmaps

- Drawback of lightmaps
  - The higher complexity of object, the greater the number of orientations exhibited by face planes
  - Cannot light up dynamic objects
- To achieve dynamic light result, influence of light is restricted (decrease the radius of light)
- Lightmaps corresponding to area within light source radius are updated only
- Typical usages – lamps, fireballs/rockets, etc.

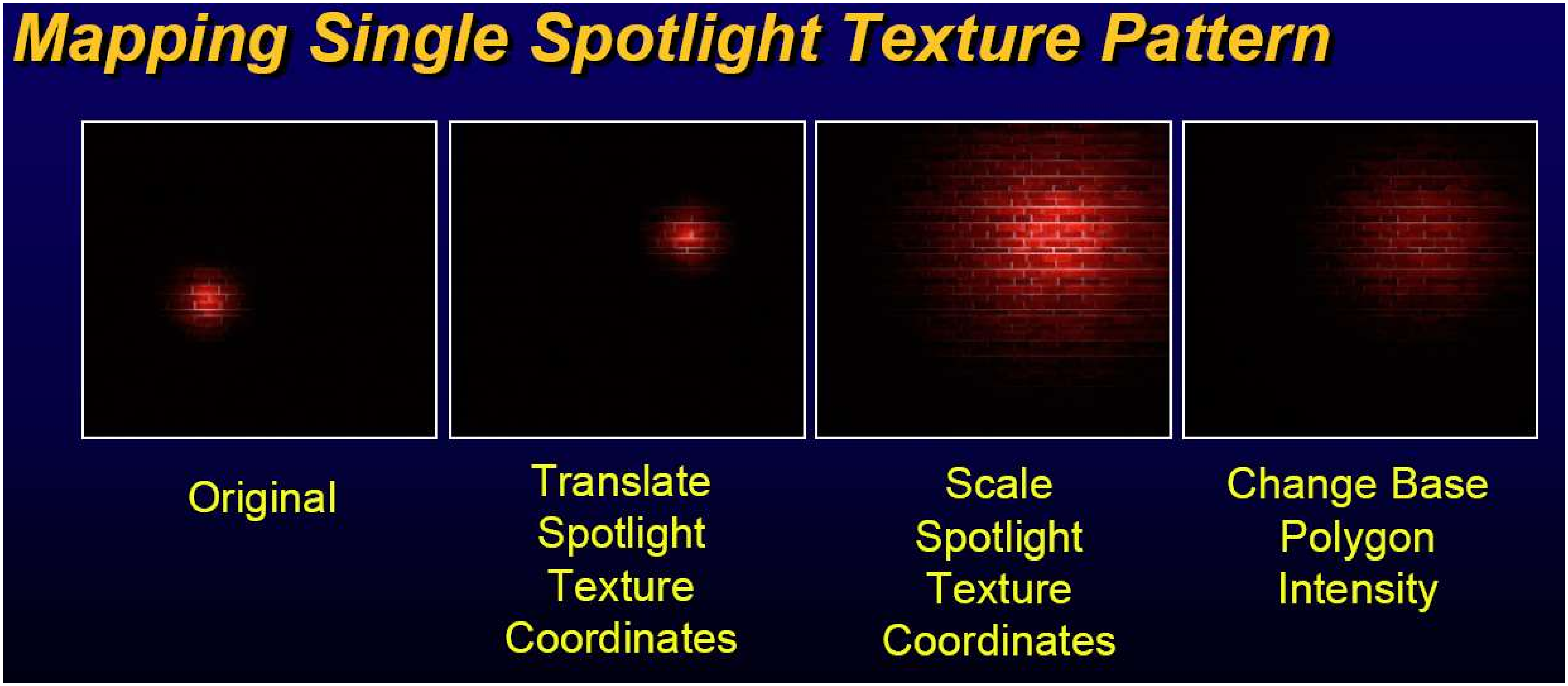
Dynamic lighting with lightmaps

- Moving (viewer centred) spotlights

- Used typically in FPS where player switch on spotlight and look into darkness
  - First use camera looking direction as a ray, and calculate intersection with scene
  - At intersection point, define a spherical light source and update the lightmaps accordingly

- Drawback
  - No longitudinal light track
  - Only single ray calculation here

Spotlight in action



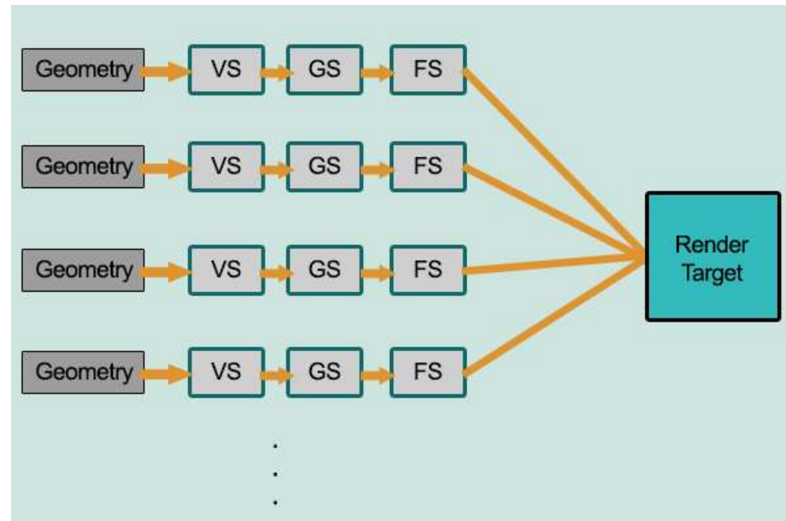
Real time Lighting

- Real time lighting on all surfaces
- Difficult to achieve as computations involved is much heavy
- Large number of light sources are possible now with next generation machines

Real time shadow calculation

Real Time Lighting

- Typical rendering loop for a scene
- For each object
  - Find all lights affecting it
    - render with lighting and material
- Or alternatively
  For each light
  - For each object
    - render with lighting
- This process will have problem when number of light sources is huge that any object will need lighting calculations many times, leading to huge loading

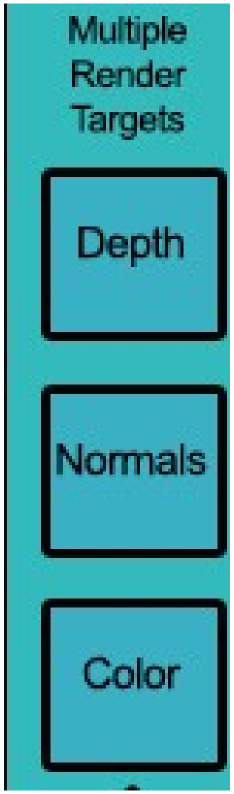


Deferred Rendering

- Separates geometry and lighting calculations
- Needs a number of buffers (G-buffers), for abstracting surface properties
- Consider Phong model
- $I = \frac{1}{a+bq+cq^2}(k_d L_d (l \cdot n) + k_s L_s (r \cdot v)^{\alpha})$ $+k_a L_a$

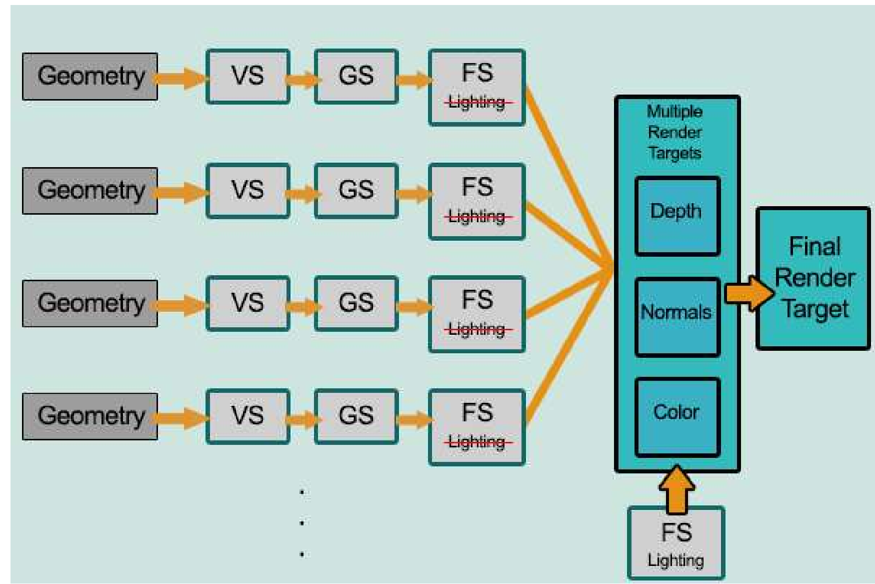
  `l`: vector from light — r: `l` reflected about n

  n: surface normal — v: vector to viewer
- With depth, normal values together with light source position, we can calculate all lighting in each pixel



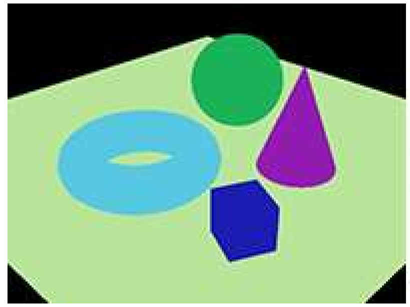
Deferred Rendering

- Rendering loop is
  For each object
  render surface properties into G-buffer
  :
  For each light
  Use G-buffer to calculate lighting in frame buffer
- Because lighting calculation is now in screen space, the calculations are limited to resolution of screen

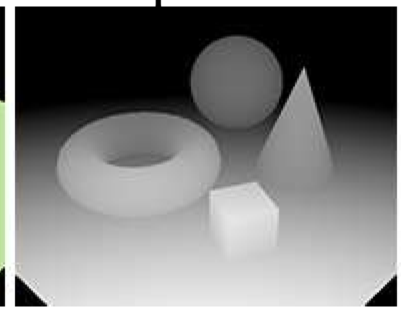


Deferred Rendering

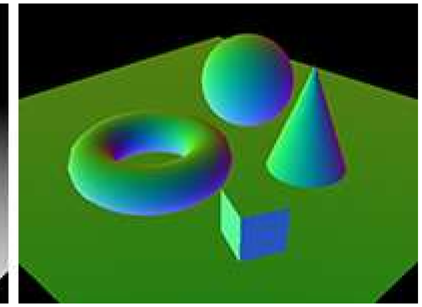
Color



depth



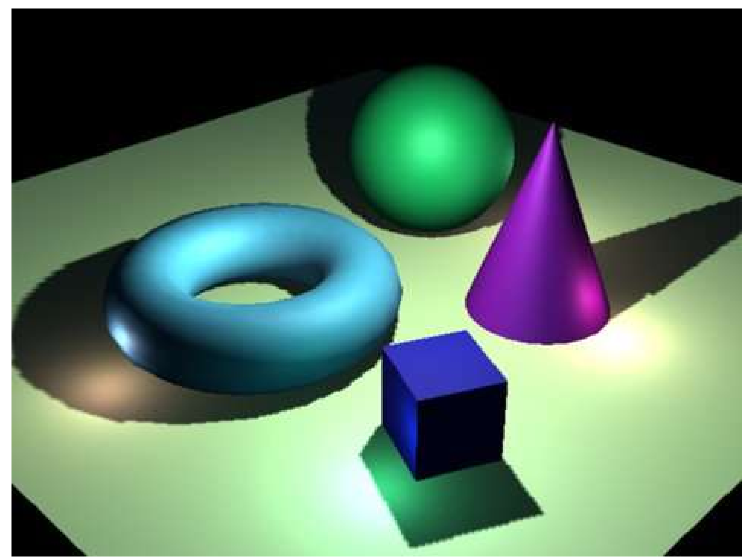
normal



- Complexity of lighting calculation is now O(screen resolution * num_lights)

- Many light sources can now be used

Rendered result



Deferred Rendering

- Seems perfect for complex lighting environment
- Drawbacks
    1. Needs big memory(3 or more G-buffers)
    2. No anti-aliasing
    3. No transparency
    4. only one material can be used

Rasterization

- Geometry converted to pixels on monitor, involve *transformation* by projection matrix
- Performed by hardware nowadays – geometry information passed in and the accelerator do the rest
- You have to roll your own for handheld & mobile

New Generation Console

- Processing power of machines are increasing at spectacular way
- Ways of programming games change accordingly
- But most of the time we are fitting/tuning our algorithms to the hardware

	Xbox Series X	PS5
CPU	8x Cores @ 3.8 GHz (3.66 GHz w/ SMT) Custom Zen 2 CPU	8x Cores @ 3.5 GHz (variable frequency)
GPU	12 TFLOPS, 52 CUs @ 1.825 GHz Custom RDNA 2 GPU	10.3 TFLOPS, 36 CUs @ 2.33 GHz (variable frequency)
Die size	360.45 mm	TBD
Processor	7nm Enhanced	TBD
Memory	16 GB GDDR6 w/ 320mb bus	16 GB GDDR6
Memory bandwidth	10GB @ 560 GB/s, 6GB @ 336 GB/s	448 GB/s

Summary

- Using game engine won't reduce tech team - remain the same size
- Nowadays the extra resources are used to more effectively distinguish the game from others
- Fitting the game design into existing engines is difficult as extensive works needed in general i.e. game engines usually will fit some particular genre