

Quaternions

A quaternion is a 4 element vector that can be used to encode any rotation in a 3D coordinate system.

$q = (w, x, y, z)$ or $q = (w, v)$ where $v = (x, y, z)$

$q = (w, v) = (\cos(\theta/2), \sin(\theta/2)r)$ where r and θ form an axis-angle rotation.

Normalise Quaternions: $w^2 + x^2 + y^2 + z^2 = 1$

Pros

Quaternions can easily be combined together, used to transform points/vectors and can be interpolated very easily. Interpolation is vital for animation, which is far more difficult with matrices.

Quaternions only use 4 floats, 12 less than 4x4 matrices.

Cons

They lack hardware support, therefore they need to be converted from matrices to them and back to matrices again.

Formulae 1

Quaternion can be converted to a matrix

If $q = (w, x, y, z)$, then

1st row - $Mq = [1-2y^2 - 2z^2 \quad 2xy+2wz \quad 2xz - 2wy \quad 0]$

2nd row - $Mq = [2xy - 2wz \quad 1-2x^2 - 2z^2 \quad 2yz + 2wx \quad 0]$

3rd row - $Mq = [2xz + 2wy \quad 2yz - 2wx \quad 1-2x^2 - 2y^2 \quad 0]$

4th row - $Mq = [0 \quad 0 \quad 0 \quad 1]$

Multiply result by $1/w^2 + x^2 + y^2 + z^2$ if q is not normalised

Can be expensive but can be simplified in code. Refer to Van Verth for more details.

Formulae 2

Quaternions can be added and scaled

Addition: $(w1, x1, y1, z1) + (w2, x2, y2, z2) = (w1 + w2, x1 + x2, y1 + y2, z1 + z2)$

Quaternions (cont)

Multiplication: $q1, q2 = (w, v) = (w1w2 - v1 \cdot v2, w1v2 + w2v1 + v2 \times v1)$

Note that X means cross product and . means dot product

Same effect as multiplying matrices, order important

This is potentially much faster than matrix multiplication

Formulae 3

Inverse of quaternion where rotation is in the opposite direction.

$q^{-1} = (w, -v)$

Quaternion must be normalised before formula is used

Much faster than matrix equivalent

Vector can be represented as quaternions. Set w to 0

i.e. Vector $p = (x, y, z) = (0, x, y, z)$ as a quaternion

Formulae 4

Rotate a vertex or vector p by a quaternion $q = (w, v)$

Rotate $q(p) = q^{-1}pq = (2w^2 - 1)p + 2(v \cdot p)v + 2w(v \times p)$

Note that X means cross product and . means dot product

Slower than matrix equivalent

Summary

Quaternions can perform similar operations to matrices with comparable performance although you need to convert to/from matrices and they can't store positioning/scaling

Therefore, there is no compelling reason to use them yet.

Emerging Tech for games

Hardware Screen res/refresh rates

Capabilities

Depth and Stencil buffer formats

Anti-aliasing

Texture Capabilities

Testing DX 10+ define min spec

Capabilities

Still need some testing to check for advance features

Consoles are largely unaffected by such matters as specs are fixed unlike PCs

Still need to check for storage size, peripherals etc.

Shader Capabilities

Shaders compiled to machine code

Shader version defines instruction set available

Higher shader versions have more instructions like for and if

Have more registers

Should provide alternate shaders for high and low spec machines

Multiple Passes

Complex material may need several passes in the shaders



Emerging Tech for games (cont)

So that one texture can be rendered through different shaders adding multiple postprocessing effects for example

Effect files for capabilities Use .fx files we can collect together shader passes and their render states into techniques

Provide a range of techniques for different hardware specifications

If any one pass in a technique fails capability testing then degrade to simpler technique

The DX effects files system makes this quite simple. Example shown in lecture slides

Geometry Shaders This shader processes primitives e.g. triangle, lines

Like vertex shader but works with multiple vertices at the same time

Operates on the output of vertex shader

Can also create or delete primitives ie output can be different to input

Emerging Tech for games (cont)

Input: Array of vertices

Output: Stream of primitives - Must be specified as a triangle strip for example. Can output any number of primitives. Example shown on lecture slides

Geometry Shader uses Distorting, animating geometry

Silhouettes

Creating extra view-dependent geometry

Particle systems without instancing

Geometry shader considerations Not needed for traditional geometry rendering methods so set gs shader to NULL

Performance of geometry shaders may be an issue for older GPUs

Stream Output stage Data output from gs can be written back into GPU memory

Very powerful DX 11 feature
Particle system can be done in 2 passes on the GPU. Pass1 - render with GPU as normal. Pass2 - Update particle positions on GPU, writing back to memory. There is no CPU intervention - efficient

Emerging Tech for games (cont)

Stream output Considerations Cannot output to same buffer that is being input from

Work around this by using double buffering

Often need multiple passes to render/update geometry

Instancing / Stream-out for Particles

Instancing Overview Instancing is a method to render many models or sprites in a single API draw call

Previously we have rendered each model one at a time

Send a list of instances with the vertex and index data

List contains what is required to render each model

Removes per-model state changes

Allows for massively increased batch sizes

Instance Buffers / State Instance data stored on GPU is instance buffer

Simplest instance buffer might contain a list of instance positions

Model defines by vertex/index data rendered once at each position in this buffer



By Jonathan_Walsh1999
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 2 of 18.

Sponsored by [ApolloPad.com](https://apollopad.com)
Everyone has a novel in them. Finish Yours!
<https://apollopad.com>

Instancing / Stream-out for Particles (cont)

State requirement for instancing can be an issue

Vertex Shaders for instancing VS often unusual when instancing, depending on what is stored in the instance buffer

Very common to store some per-instance data and randomise other elements

Instance Buffer Data Can store more than just position in an instance buffer to give each instance a different look: Rotation, scale or store entire world matrix per-instance

Can also store more unusual data: Seed value to randomise each instance or entity/particle data to allow the model to be updated on the GPU using stream-out

CPU / GPU Instancing Simple instancing is processed using both CPU and GPU. GPU render instances and CPU update instances

Instance buffer must be made available to both CPU and GPU

Instancing / Stream-out for Particles (cont)

Space is reserved for instance data in both CPU and GPU memory

Constant copying of instance buffer between GPU and CPU means performance is lower than normal

This is why we might not want to store a world matrix for each instance. Instead the data is often compressed

Implies VS may have to do additional work to derive the full instance data

Using Instancing Instancing suits the rendering of large numbers of similar models. ie trees, armies

Example: Particle Systems Particles are all similar, often camera-facing sprites

Particle systems are an ideal candidate for instancing

Each particle system stores rendering data such as position, rotation, scale, colour, alpha

Each particle requires data to update its position/rotation each frame

Instancing / Stream-out for Particles (cont)

Particles are spawned from emitters

Particles have a life time after which they die

There may be attractors, repulsors and other features added for system complexity/flexibility

Approach: Store render data in instance buffer, store update data, update particles using CPU and then copy entire buffer to GPU, render particles in one batch using instancing, much faster but still requires CPU/GPU copy

Sprite-based particle systems Smart approach for camera facing sprite particles however this method can't be used if the particles are models

Advanced Instancing Instancing can look poor due to lack of variety

Complex instancing techniques store more states e.g. animation data, texture offsets, material settings



Instancing / Stream-out for Particles (cont)

Able to render models in different poses, with different-textures and material tweaks. Good for vegetation, crowds etc.

More complex shaders can help here

Latest GPUs deal well with this kind of shader

Particles without CPU/GPU copy Instancing can be slow due to the CPU update/copy

One simple workaround is to avoid updates.

Drawback is that it is inflexible as paths are always the same. e.g. fountain can't be affected by wind

GPU stream-out for particle update DX 10 supports stream output. Allows GPU to output vertex data back into a vertex buffer instead of sending it on for rendering.

Using stream output the GPU can be used to update particles for entities position, rotation etc

Both render and update data is stored GPU only

Instancing / Stream-out for Particles (cont)

Typically we render the models twice. Pass1: Render models using instancing or similar. Pass2: Update models with stream-out - no actual rendering

Stream output considerations Reads from GPU buffer and writes back to one but can't output to same buffer that is being input from. Work around this by double buffering

Stream-out allows GPU only entities which is especially effective for particles.

Works especially well with the sprite-based particles technique

DX 11 - New Features

New Features DX 11 was introduced with Win7

Features include multithreading, tessellation, compute shaders, shader Model 5.0 and high quality texture compression formats.

DX10 DX11 Differences Nearly everything DX10 works with minimal change in DX11

DX 11 - New Features (cont)

Device pointer has been split in two. Device pointer for overall control and context pointer for each thread

.fx not in the provided libraries

DX maths libraries not in 11

No font support

Few other minor changes

Pipeline Get two programmable stage: hull and domain shaders

One fixed stage in between: Tessellation

All three must be used together for tessellation otherwise disabled

Tessellation Input geometry made of patches and control points.

Vertex shader processes each control point

Hull shader also processes each control point but can access all points for a patch. Used for specific transforms.

Hull shader has an associated patch constant function which is called once per patch

Tessellation stage tessellates the patch as required

C

By [Jonathan_Walsh1999](https://cheatography.com/jonathan-walsh1999/)
cheatography.com/jonathan-walsh1999/

Not published yet.

Last updated 11th May, 2020.

Page 4 of 18.

Sponsored by [ApolloPad.com](https://apollopod.com)

Everyone has a novel in them. Finish

Yours!

<https://apollopod.com>

DX 11 - New Features (cont)

Domain shader takes the generic tessellation and control points and creates the final vertices

Patches/control points A Patch is a line, triangle or quad which is bent or shaped by some number of control points

DX does not specify the available patch types

This is potentially a huge change for game asset creation

Hull shader Gets access to all control points for a single patch and can process them in any way

Output: Final control points used to shape the patch. MAY output greater or fewer points if necessary

Can be used for advanced purposes like approximating complex input splines using simpler output splines. providing per control point info to help the patch constant

Patch Constant Function Called once per patch - decides how much to tessellate each patch

DX 11 - New Features (cont)

Access input control points and the hull shader output control points as array to do its job

Tessellation Stage Uses factors specified in the patch

Divides up a unit square, triangle or line based on the factors

works in a generic 0->1 space

Several fixed algorithms are available for the tessellation

Domain Shader Takes control points output from hull shader and the generic vertices output from the tessellation stage

Combine to create final tessellation for the scene

Exactly what this involves depends on the patch type.

Distance / Density Variation Common to vary amount of tessellation based on the geometry distance

Distance variation is simpler

Density variation needs pre-processing

Water-tight patch seams As tessellation is varied there are problems with patch seams. - cracks in geometry appear

DX 11 - New Features (cont)

That is why we can control the edge tessellation separately to ensure all edges have the same tessellation factor.

Displacement Mapping Adjust height of vertices

Effectively this parallax mapping done properly

Result has correct silhouettes and no visual problems

Technical Issues Tessellation has performance implications

Displacement mapping brings more seam issues

Models must be designed with displacement in mind

Stereoscopic Rendering

Depth Perception - 2D Number of depth cues in a 2D image/video

Pos and perspective

Known sizes of objects

Visible detail

Motion Parallax

Shadows and lighting

Occlusion - nearer objects hide further ones

Atmospheric blurring - distance fog

None of these require 2 eyes just monocular vision

Binocular Vision We gain additional cues from having 2 eyes



Stereoscopic Rendering (cont)

Image in each eye is different

Brain resolves into one image with depth

Not sure if this will come up in exam so only covered briefly

Animation: Interpolation

Interpolation is where a calculation is made to decipher a transform between 2 control transformations of a model

An animation is stored as a sequence of key frames (or transforms).

In order to get the frames in between the key frames, interpolation is used

Interpolation occurs in alpha blening and skinning

Linear Interpolation (Lerp)

Interpolation between two mathematical elements (could be points) P0 and P1

$$P(t) = P0(1-t) + P1t$$

Where t is typically in the range [0, 1] and the start and end elements are P0 and P1 respectively.

The interpolated point will be on a straight line in between P0 and P1, hence linear interpolation

Normalised Lerp (Nlerp)

Can use Linear Interpolation for transformations including translations, scaling and rotations, however, the results for rotations is not correct, resulting in unwanted scaling. Therefore, Nlerp or normalised Lerp is required for rotation.

This works however, the angles can still be inaccurate. Can use Nlerp for rotations if the overall rotation is small enough.

Spherical Linear Interpolation (Slerp)

Linear interpolation of angles is same as linear interpolation of an arc on a sphere.

Formula different from linear interpolation (Lerp)

Animation: Interpolation (cont)

$$\text{slerp}(P1, P2, t) = P1(P1^{-1}P2)^t$$

More suited for larger rotation as it calculates the correct interpolated rotation

Slerp for Matrices: Substitute the matrices into the formula. Required to raise the matrix to the power with t. This means that we need to convert the matrix to an axis-angle format then calculate θ^t then convert back.

This is very expensive

Slerp for Quaternions: The only thing that makes it make expensive is the sine function. There can be accuracy problems for small theta, but more useable than the matrix version

$$\text{Quaternion formula: } \text{slerp}(P1, P2, t) = (\sin((1-t)\theta)P1 + \sin(t\theta)P2) / \sin(\theta)$$

Summary

Can use Lerp for positioning and scaling

For small rotations use nLerp

For larger rotations use Slerp

Rotations should be stored as quaternions if interpolation is involved as matrices are expensive

Animation: Practicalities

Matrices are not good at animations as they are performance heavy use far too much storage, so quaternions should be used instead

We can decompose the transformation into rotation, translation, scale etc., using vectors for translation and scale and quaternions for rotation

Spatial Partitioning

Spatial Partitioning is any scheme that divides the game world into smaller spaces

Needed for larger scale games

Spatial Partitioning (cont)

Problems with Large Games Complex games can contain millions of instances

The majority of instances are likely to be far from the player

We would like to cull these instances instead

Simple Culling Methods Can cull entity instances against the viewing frustum. This is the volume of space visible from the camera, which is a cone with its head cut off.

Check each instance against each of the 6 planes defining the frustum or more simply rejecting those behind the camera near clip plane

Use bounding volumes and simple maths like boxes or spheres

Rationale for Spatial Partitioning Culling instance one-by-one is not the best approach for very complex environments. There are too many instances to even consider in one frame.

Spatial Partitioning (cont)

Need to reformulate problem and don't process non-visible instances at all

Partitions can be seen as chunks of space and instead identify which partitions are invisible allowing use to accept or reject large groups of instances at once.

Simple Example Most space partitioning schemes use some form of graph to subdivide the world where each node represents a space. Shape of the spaces vary by scheme. The edges represent how the spaces are related or connected.

One example shows a very basic partition/graph demonstrating how areas in the scene are connected and how a group of instances can be rejected by one check. (Refer to lecture slides for diagram)

Level Division Space partitions are not just for visibility checks

Spatial Partitioning (cont)

This can help in a variety of non-rendering situations.

For example a game can be partitioned into levels. Another example could be loading or releasing resources when moving between different partitions. Or having new pp or lighting effects or changing music etc.

Game Logic Space partitions can also help with game logic

For example a race track can be split up into sectors where only the current and neighbouring sectors enable AI physics and rendering because AI race cars which are far away don't need physics etc. because you can't see them.

Spatial Partitioning (cont)

These sectors can also simplify lap processing which can include distance covered, telemetrics or detecting whether you are going the wrong way around a race track.

Visibility/Audibility Partitions can be used to determine whether you can hear sound past a concrete wall for example.

Potentially Visible Sets (PVS) Each node in a space partition has a potentially visible set (PVS)

These are the nodes that can in some be seen from that node. For example, you can see the living from the hallway because you can see through an open door. (Diagram shown in lecture slides)

PVS can be pre-calculated and stored with each node. This indicates which other nodes to render when in that node.

Generating PVS A PVS scheme is conceptually simple



Spatial Partitioning (cont)

However, generating the PVS for each node is non-trivial

Possible approaches include using brute force, which considers many different camera positions. This can be slow and result in possible errors. You can manually create PVS. This can only be possible for simpler graphs and is error prone. Finally mathematical/geometric approaches can be used, which are complex and often have limitations

PVS Limitations PVS does not consider dynamic geometry. For example if you have a level that has a door which opens then the door must be considered as open for PVS

Potentially visible sets must be conservative. For example, a node visible from only a tiny portion of the current node would need to be entirely visible

Spatial Partitioning (cont)

So whilst efficient to execute, PVS systems are not ideally effective in node rejection.

PVS Use PVS system is not space partitioning scheme as such

PVS can be added to any space partition graph regardless of scheme used

Used as a quick way to reduce the number of nodes under consideration

Portal Systems A Portal system is a method that concentrates on the graph edges

Spaces in such a system are connected through portals. A portal is typically a natural opening such as a door or window

Portals allow us to reject other nodes based on the camera view

Basic Portal usage Identify which node the camera is in

Identify whether each of the node's portals are visible in the viewport

Spatial Partitioning (cont)

Now we know the nodes connected through the visible portals are also visible

Refinements When a visible portal is found store its viewport dimensions (2D rectangle)

Clip portals in the connected node against this smaller area. Reject obscured nodes

Watch out for multiple portals leading to same nodes. We don't want to render nodes twice.

Portal Pros Cheap and simple implement

Effective for indoor geometry

Portals can handle dynamic geometry (unlike PVS)

Each portal with 2 sides don't need to be in the same place.

Portal Cons Can be tricky to know which node a particular point is in



By [Jonathan_Walsh1999](https://cheatography.com/jonathan-walsh1999/)
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 8 of 18.

Sponsored by [ApolloPad.com](https://apollopod.com)
Everyone has a novel in them. Finish Yours!
<https://apollopod.com>

Spatial Partitioning (cont)

Need to know which node the camera is in to start the algorithm. e.g. what if a camera travels through a wall or teleports?

Portals are of little use for open areas

Not easy to automatically generate portals from arbitrary geometry

Grids as Spatial Partitions Can collect local entities for visibility culling like AI

Can be used to map terrain (Height/influence maps)

Can be extended to 3D

Disadvantages to Grids as SP May have many empty nodes, wasting memory, reducing cache efficiency

Choice of partition size tricky - too small gives many empty nodes, too large and culling etc. is ineffective

Mapping a Grid to the World A grid is an integer indexed structure for a rectangle of world space

Spatial Partitioning (cont)

Need to map between world space coords and grid indices

Conversions for X dimension are (Y similar):

$$\text{GridX} = (\text{int})(\text{GridWidth} * (\text{WorldX} - \text{MinX}) / (\text{MaxX} - \text{MinX}))$$

$$\text{WorldX} = \text{Min} + (\text{float})\text{GridX} * (\text{MaxX} - \text{MinX}) / \text{GridWidth}$$

2nd formular gives bottom-left of grid square

Quadrees / Qctrees Quadrees / Qctrees are hierarchical partition systems which use a tree structure to represent an area/volume of space.

Use specific division scheme

Quadrees are in 2D, Octrees in 3D

Creating a Quadtree Root node is entire space

Divide into four equal quadrant

Repeat division with each quadrant

Until some condition is met - max depth, empty node etc.

Location in a Quadtree Easy to find which node point is in

Can be optimised

Can use bitwise integer math

Spatial Partitioning (cont)

Quadrees for visibility culling Use for frustum culling

Viewing frustum is 6 planes

Test if a node is visible

Quadtree Problem Entities aren't points

May overlap a node boundary

Entity needs to be in a larger parent node

Worst case: entities overlaps origin and does not fit in any node except root and will never be culled

Hot-spots like this all the way around the boundaries of larger nodes.

Solution Loose Quadrees

Have nodes overlap

Entities will then fit in original node area

Few changes to algorithm - increase node size when inserting entities and when doing frustum culling

Removes hotspot problem



By [Jonathan_Walsh1999](https://cheatography.com/jonathan-walsh1999/)
cheatography.com/jonathan-walsh1999/

Not published yet.
 Last updated 11th May, 2020.
 Page 9 of 18.

Sponsored by [ApolloPad.com](https://apollopod.com)
 Everyone has a novel in them. Finish Yours!
<https://apollopod.com>

Spatial Partitioning (cont)

At the expense of larger nodes at the same level

Quadtrees for Collision Detection Saw intersection of viewing frustum with quadtree

Easy to find intersection of other primitives - sphere, cuboids, rays etc.

Basis for collision detection/ray casting/particle systems

Can help if we add adjacency info to the tree

Binary Space Partitioning (BSP) Hierarchical division of space and uses another tree structure. This one represents all space

Partitions are separated by lines in 2D or planes in 3D

Recursively divide each partition into 2 smaller ones

Creates a binary tree

Creating a BSP Repeatedly divide space in 2

Spatial Partitioning (cont)

Stop when max x elements in each partition. Partitions are small enough. tree reaches certain depth and choice depends on application

Locating a Point in a BSP Given a point, each to find which partition it is in. Start at root of tree

Look at example in lecture slides

BSP for solid/hollow spaces Can use the polygons in the scene as the division planes. Choose a polygon as a plane and polygons crossing the planes are split

BSP splits space into hollow/solid volumes

All polygons/entities places in hollow ones

BSP / Brush modelling Traditional style of BSP used for FPS games

In conjunction with PVS

Can also be used to render partitions in a strict back to front order

Spatial Partitioning (cont)

Lends itself to a unique form of 3D modelling called brush modelling. You start with a entirely solid world, cut out primitives, entities paces in hollowed out areas. This is like digging out the level.

BSP Pros and Cons +BSP trees are a well established technique

+Used for rendering/collision/ray-tracing

+Can be generated automatically

+Fully Classify space

-Need good algorithm to choose dividing planes

-Hollow/solid BSP generates extra polygons due to splitting

Deferred Rendering

Forward Rendering Name for the method of rendering we have used in all material so far

Render geometry and light effects on the geometry in single pass

Cost = numObjects x NumLights - Get's very expensive



Deferred Rendering (cont)

Forward rendering can be effective but need a slow uber-shader or lots of shaders and batch problems

Doesn't work well with lots of lights in one place

Deferred Rendering Decouples geometry from lighting

Splits the rendering process into 2 stages

Cost = NumObject + NumLights - Much cheaper

G-Buffer Render geometry to g-buffer, which is several textures holding geometry and surface data

Example: Texture1: Diffuse Colour Texture2: WorldPosition Texture3: WorldNormal

Pixel shader can render to several render targets at the same time, so can build three textures all in one pass with a special pixel shader

MRT = Multiple Render Target

Data in g-buffer is anything we need to calculate lit version of the scene

Deferred Rendering (cont)

Large g-buffer results in major performance drain - memory access is slow...

So data compression in the g-buffer is common ie store x and y of normal together with a single bit for direction

Lighting Volumes G-buffer is not displayed

Render actual scene by going through each light and rendering it's effect on the geometry

Point light lights up a sphere around itself. Render the sphere around the point light. For each pixel find if it is actually lit up. Use data in g-buffer to calculate amount of light. Do this for every light and accumulate = rendered scene

Same concept for spotlights

Don't need high-poly spheres or cones

Examples shown in lecture

Deferred - Pros and Cons +Lights become cheap to render

+No need for complex partitioning

+Shaders become simpler - less of them

Deferred Rendering (cont)

+Better batching performance

+G-buffer data can be reused for Post-Processing

-Huge g-buffer can be a slow down

-G-Buffer compression to counter this reduces material flexibility

-Transparent objects don't work, must be rendered separately

-MSAA becomes very difficult due to g-buffer

-Not actually particularly useful in some scenes(daylight)

More advanced techniques are getting very complex

Optimisation for Games

Optimisation Tradeoffs

Reducing memory use can decrease speed

Increased speed might be at the expense of memory

When not to optimise

Never optimise code unless you are sure that it affects performance

Optimisations usually harm readability/maintainability of code

Can reduce functionality

Can make architecture less flexible

Performance Analysis

Generally, 90% of processor time is spent on just 10% of code

Need to identify the 10% to optimise effectively

Tools can be used to analyse performance of code during run-time



Optimisation for Games (cont)

Performance Analysis Tools

- Simple timing functions
- Profiler - Reports on time spent in different functions
- Specialist tools like VTune, PTU, PerfKit, PerfHUD etc.

Compiler Optimisations

Compilers can perform some optimisations
Optimisations can be enabled using release mode in visual studio.

Basic Language Optimisations

- Loop Unrolling - Does not loop through indices, just duplicated lines of code instead
- Remove constant calculations by using a variable outside a loop for example
- Change ording of conditions, like OR for example. Put simple condition first
- Pass by reference not copy
- Use early return within functions whenever possible
- Inline functions - stores functions in cache but can be ignored by compiler
- Break code into smaller steps. For example, don't have calculations inside if statements. Does not directly lead to optimisations but can help compiler optimise.
- Try programming in assembly, although it would be very complex and compilers would probably do a better job.

Data Structure Choices

- Static structures like fixed arrays might improve performance over dynamic ones
- Only choose data structures that suit your needs, nothing more

Algorithmic Improvements

- Can multiply by 0.5 rather than dividing by 2
- Reduce nesting of loops - don't go deeper than 3

Optimisation for Games (cont)

- Reduce range of loop counters
- Sort data into more convenient orders
- Cluster similar cases into one
- Reduce maths operations
- Pre-calculate formulae using look-up tables
- Remove code completely!

Alpha Sorting and Soft Particles

Alpha Sorting Problems Attractive blending technique but causes sorting issues

Problem is depth buffer ignores transparency

Avoid problem by drawing polygons back to front.

Run-time Depth Sorting If all polygons face camera ie particle system then you can sort polygons based on camera-space z distance

Issues arise with this based on example shown on slides with the lines

To solve this assume polygons don't intersect

Then given 2 polygons one of them will be entirely on one side of the plane of the other

Identify this polygon and see if it is on the side nearer the camera or not

Alpha Sorting and Soft Particles (cont)

First step is to get a face normal for each polygon

Join either point of polygn 2 to eachh of the points polygon 1. Calculate dot products of these with normal of polygon 2. Results all +ve : poly 1 is nearer. Results all -ve: poly 1 further. Results mixed: poly1 is split by place of poly 2. So repeat test the other way around. If split both ways then the polygons are intersecting. Refer to slides for diagrams etc.

Run-time sorting practicalities Must ensure this sorting is efficient as possible. so sort pointer to polygon not polygon data itself

In practice, another technique alpha-to-coverage is often used as an alternative.

Hard Flat particles Alpha blending is as useful as other blending methods once the polygons are sorted

However all blending methods exhibit hard edges if they intersect other polygons



Alpha Sorting and Soft Particles (cont)

Particularly large particles like smoke indoors

Soft Particles To improve further we can compare depth of particle with depth already in buffer and then fade pixels out when the distance is small. - Adjust alpha toward 0

Depth-Soft Particles This method can be combined with the depth particles idea presented earlier

We must do some detailed work with depth buffer but almost completely removes hard edges where alpha particles intersect solid objects.

Further Possibilities Can explore volumetric particles - consider the volume of particle that camera is looking through.

Linear Dynamics and Particle based Physics

Particle System Basics Data: Position, velocity, possibly mass

Linear Dynamics and Particle based Physics (cont)

Particle velocity must change or it will only move in a straight line. Change in velocity is called acceleration. Acceleration caused by forces on particle. Gravity is common force.

Particle Update $F=ma$

Use above formula to update particle each frame

Diagram shown in lecture slides

Aprox. in this update This ibasic physics of forces, acclerations and velocities doesn't just apply to particles. Starting point for modeling physics too.

Problem: Approach is only an approx. we only update things once per frame. Assumes vecocity was constant over entire time period of rame. This is wrong - forces/acceleration will change gradually throughout frame. Whereas our simple approach changes the velocity isntantly to a new value each frame.

Linear Dynamics and Particle based Physics (cont)

Example of this is when you have a particle following an orbit around an object. Over time the particle will move further away from the object it is orbitting. This is down to approximations and is wrong.

Initial Value problems Updating particle pos is an example of an initial value problem. We know the value of an equation at an initial point in time. Want ot calculate value at some furutre point in time.

In this case we know pos and velocity from this frmae. Want to know position and veclicity for next frame. The simple but flawed method just shown is one way of solving an initial value problem. Will present others with better accuracy.

Formal Definition Function which changes over time: $p(t)$

Initial position/veclicity: p_0 (where $t = 0$)

Time period: h

Value next frame: $p(t_0 + h)$

Need derivatives: $p'(t), p''(t)$



By Jonathan_Walsh1999
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 13 of 18.

Sponsored by [ApolloPad.com](https://apollopad.com)
Everyone has a novel in them. Finish Yours!
<https://apollopad.com>

Linear Dynamics and Particle based Physics (cont)

1st derivative of pos = velocity.
2nd = acceleration

Euler's Method Taylor series is a representation of a function based on the derivatives at a single point (int time)

$$p(t + h) = p(t) + hp'(t) + \frac{h^2}{2}p''(t) + \frac{h^3}{3!}p'''(t) + \dots + \frac{h^n}{n!}p^{(n)} + \dots$$

Arranged here to suit our problem

p is pos, p' velocity, p'' acceleration, p''' acceleration of acceleration

As h is smaller approx is more accurate

IT is an infinite series - cannot be completely calculated

Euler's Method uses just the 1st two terms in the series and assumes the rest are small enough to ignore.

Translation into games terms:
posNextFrame = currentPos + frameTime * currentVelocity

velocityNextFrame = currentVelocity + frameTime * currentAccel

Linear Dynamics and Particle based Physics (cont)

This is exactly the method presented earlier for updating particles in a particle system. Not ideal, terms are ignored (not always small). Still widely acceptable when accuracy isn't required.

Mid-point Method Problem with Euler's method is that velocity and acceleration are taken at the start of the frame.

The mid-point method takes them half way through the frame.

This has better accuracy than Euler's method but not perfect as half-way values are themselves approx.

Basic Verlet Method Less reliant on velocity

Can be restrictive because of that

OK for particle systems if only concerned with position

Most basic method: uses pos from the current and previous frame and uses current acceleration

formula: $y(t+h) = 2y(t) - y(t-h) + h^2y''(t)$

Linear Dynamics and Particle based Physics (cont)

$$\text{posNextFrame} = 2 * \text{currentPos} - \text{posLastFrame} + (\text{frame time})^2 * \text{currentAcceleration}$$

Has similar accuracy to mid-point method

Particle Physics - Springs Forces involved: Gravity, spring compression and spring stretch

Considers particle mass

Spring Forces Force exerted by spring is from Hooke's Law: $F = -kx$

x = displacement for spring's equilibrium pos

k = spring coefficient (stiffness)

Practicalities Real life systems slow down with friction

Instead of friction we will damp the motion

Damping force: $F_d = -cv$

v = velocity

c = damping coefficient - works against current velocity range: 0-1

Uses Can model rope, cloth and jelly-like objects

Different Connectors Can use new connector type such as elastic, rods and string



Linear Dynamics and Particle based Physics (cont)

Key difference introduced: Some types behave differently when stretched and compressed, some are constrained, some don't exert forces at some times.

Constraints Rods and strings have constraints. Rods must always be same length and string cannot be longer than original length

Mathematical Approaches Each constraint can be written as an equation illustrating then fixed length between particles such as: $|p_i - p_j|^2 - L_{ij}^2 = 0$

p is particle pos and L is fixed length of connector

Several constraints we have several equations

Known as a system of linear equations

Solving Constraints Of the various mathematical solutions most have a similar repeated iterative approach. Examples shown in slides

Advanced Graphics: Scene Post-Processing

Front/back buffers Visible viewport can be called front buffer

A 2nd off-screen back buffer is the usual render target

Advanced Graphics: Scene Post-Processing (cont)

After frame rendering the back buffer is copied to the front buffer
This is a form of double-buffering

Swap method s/c-hains Methods to get the back buffer content to the front buffer involve a simple copy were the back buffer is discarded or the 2 buffers are swapped which is useful if we want to keep the last frame

Can have more than one back buffer. This is known as triple-buffering

Improved concurrency with GPU

Multiple back buffers must use the swap method which is called a swap chain

VSync or Not Copy/swap is fast operation

Can perform it during the monitor's vertical sync

If you do this though the FPS will be tied to monitor refresh rate

Advanced Graphics: Scene Post-Processing (cont)

Alternatively can copy to front buffer immediately. - May see tearing

Alternative Render Targets Not necessary to render to a back buffer

We can render to a texture or to a specially created render target

Can create explicit render targets or render to multiple render targets

Scene Post-Processing Assume we render the entire scene to an intermediate texture

Can then copy it to back buffer to be presented to the viewport but we can also perform additional image processing during this copy

The copy process is effectively another rendering pass so the look of the scene is altered through pixel shader

This is full-screen post-processing

Multiple Passes Can post-process in multiple passes

Render Targets

C

By Jonathan_Walsh1999
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 15 of 18.

Sponsored by [ApolloPad.com](https://apollopad.com)
Everyone has a novel in them. Finish Yours!
<https://apollopad.com>

Advanced Graphics: Scene Post-Processing (cont)

The textures used do not have to all be the same size so that you can scale down and back up for blur for example

Can make complex sequences of post processing like bloom.

Don't need to talk any more about Post Processing - Should be confident from assignment

Water Rendering

Visual Aspects of Water Reflection, refraction, fresnel effect, kught extinction, surface deformation, foam/spray/c-austics and underwater effects

Reflection Water behaves to some degree like a mirror

Perfectly still water presents a perfect reflection

Surface deformation presents practical difficulties as the normals vary

Reflection Practicalities Can be dynamic, movement in scene is reflected

Or static - Just skybox reflected

Water Rendering (cont)

Static case - Cube mapping works effectively, reflect ray from camera off the surface normal and into a cube, hlsl support for cube-mapping makes this simple, works without difficulty with varying normals

Dynamic reflections - cube mapping not effective so reflect the camera in the plane of the water, render the scene from this reflected camera into a texture, draw the water surface mapped with this reflection texture

Varying normal can be simulated by offsetting which part of the reflection texture sampled

Not a fully robust solution. Reflections might come from parts of the scene that were not rendered in the reflection texture. Approach only works perfectly for completely flat water

Water Rendering (cont)

Alternative approach is to use ray-tracing or similar

Self-Reflection If the water surface is choppy enough it may reflect other parts of the water

Reflection and refraction require multi-pass approaches to do properly however don't need to do it properly in most cases

Static cube mapping: Lower half of cube map not really needed so draw the upper half reflected

Dynamic reflected camera: render the water in the reflection texture using static cube mapping

Refraction Where light crosses the interface between 2 different materials it bends

Amount of bend is given by Snell's Law

Depends on: Angle of incidence, refractive indexes, vacuum has a refractive index of 1, clean water is 1.33

$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$

C

By Jonathan_Walsh1999
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 16 of 18.

Sponsored by [ApolloPad.com](https://apollopad.com)
Everyone has a novel in them. Finish Yours!
<https://apollopad.com>

Water Rendering (cont)

Refraction in Water When looking into water, light coming from under the water is bent and the scene at the water surface appears shifted and distorted

Amount of shift/distortion depends on: angle at which we view the surface, variations in the surface shape - waves ripples, both of these vary per pixel

Refraction - Practicalities Refraction typically rendered in the manner of a post processing effect - similar to distorted glass

Process - Underwater parts of scene rendered to texture, water surface is rendered and this texture is applied, distortion is applied to UVs

Fully robust system would be complex

Combining reflection and Refraction Both involve rendering scene to texture

Water Rendering (cont)

In practice: Create 2 textures, render sabove water scene(reflected) to 1 and the below water scene to the other. Clip each of these scenes at the water surface

Render water surface blending reflection and refraction textures

Blending amount depends on viewing angle

Fresnel Effect To do with viewing angle and blending of textures

Effect depends on the material involved

$$F = F_0 + (1 - F_0)(1 - N \cdot C)^5$$

$$F_0 = ((n_1 - n_2)/(n_1 + n_2))^2$$

n_1, n_2 are the refractive indexes of the material

N = surface normal C = Normal to the camera

Water Rendering (cont)

F gives the proportion of reflected light coming from the surface, the remainder comes from refraction. e.g. if $F = 0.3$ at a point on the surface. Point emits 30% reflected light and 70% refracted light

Fresnel formula calculated in pixel shader giving a blending ratio for the reflection and refraction textures

Light Extinction Light attenuates in water as well as air

The effect in water is much stronger though

Practicalities Effects refracted light only

Need to know how far light has travelled

C

By Jonathan_Walsh1999
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 17 of 18.

Sponsored by [ApolloPad.com](https://apollopad.com)
Everyone has a novel in them. Finish Yours!
<https://apollopad.com>

Water Rendering (cont)

Several approaches can be used: e.g. render water surface only to texture, store only its world space distance from camera, when re-rendering refraction texture subtract the distance of each underwater pixel from the water surface distance at the same point. Gives distance the light travels through water to surface. Linearly blend RGB components based on this distance and the extinction distances given. Water surface distance texture created in the 1st step can also be used to do the above/below water clipping

For surface normals you can animate normal maps to get a wave or ripple effect etc.

Refer to lecture for more detail

C

By [Jonathan_Walsh1999](#)
cheatography.com/jonathan-walsh1999/

Not published yet.
Last updated 11th May, 2020.
Page 18 of 18.

Sponsored by [ApolloPad.com](#)
Everyone has a novel in them. Finish
Yours!
<https://apollopad.com>