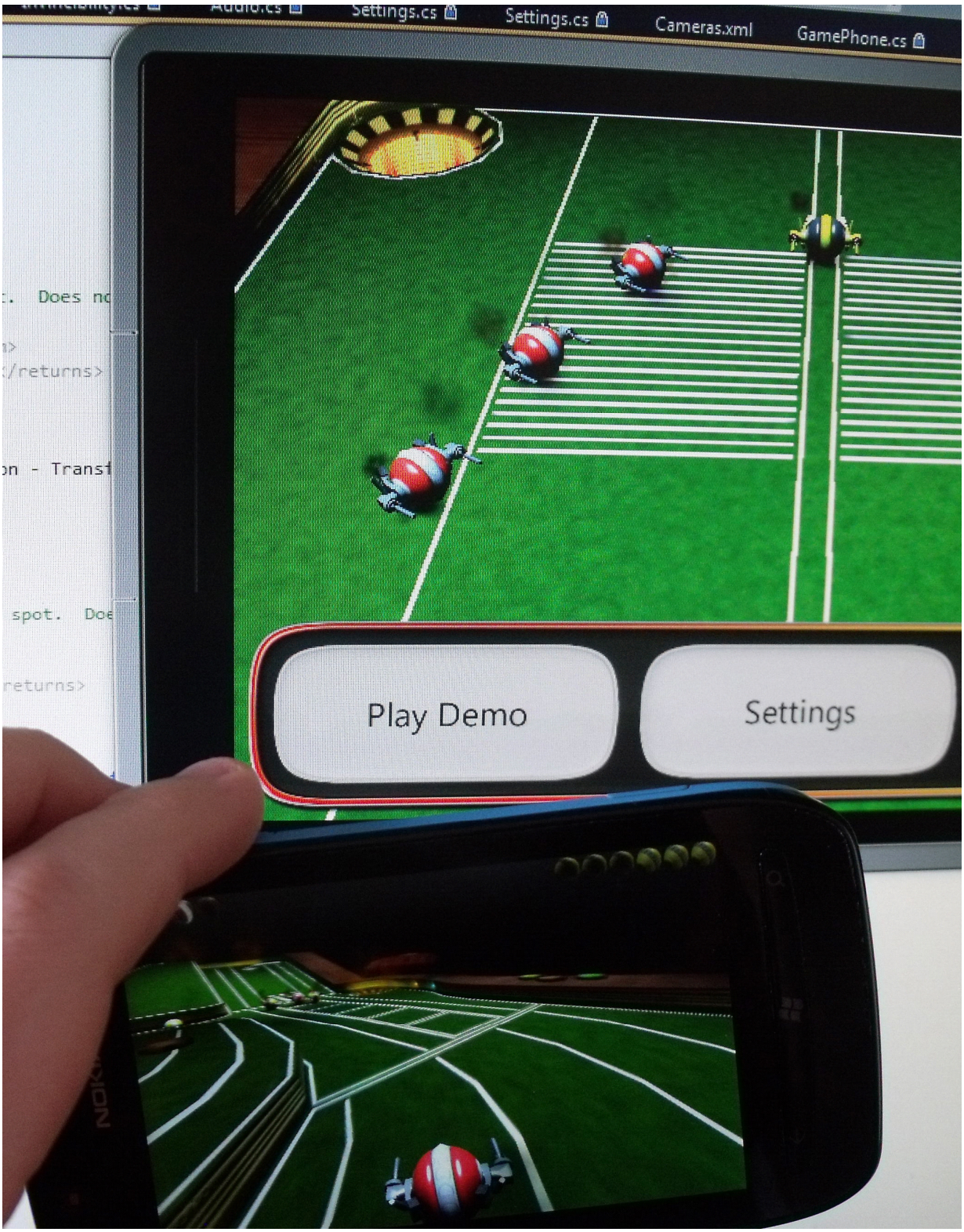


Code Reuse In Games - James Wild

2 - Abstract

Over time, software in general is becoming more complex. The core executable for UDK 08/2011 Windows 64-bit was 55,183kb [1], for the game engine alone. This includes no assets, not even scripts that run most of the gameplay in the bundled UT3 demo [2]. Other executables, such as Skyrim's core executable, is a less at only 17,063kb [3]. However, neither of these figures includes the mass of .dlls that are loaded into memory alongside. And these figures only come into perspective when one considers how small a game of yesteryear, including all assets, is. To continue to use the Elder Scrolls series as an example, Arena, the first game, is only 8.8Mb [4] including all content despite containing a world twice the size of Great Britain. (approx. 9011Kb) Cartridge-based formats such as the Game Boy or Nintendo 64 were further miracles of minimal disk space; Game Boy cartridges were only 256kbits[5] in size.

This is not, however, much of a problem in itself with modern multi-layer DVDs offering 8.5 GB (8965324.8kb) [6] of storage. What it does indicate, however, is that with time code complexity is growing at an exponential rate. Whereas in a game like Pokémon rendering code was a matter of copying tiny bitmaps to tile memory and setting registers to display them, likely hand written in assembler, a modern renderer is a massive codebase involving multiple simultaneous code paths [7] that load content while the game plays and structure the world in such a way that it can be rapidly analyzed and drawn.

Employees had to be paid to write every line of code, and with every line of code the complexity of the codebase as a whole increased exponentially. These sections of code interact with one another and sometimes an unconsidered edge case arises (what if the player drowns during a level section transition, for example) and bugs occur. Bugs are not as much of a problem in practice now as they were back when games were less complex. With collective years of experience, the quality of programmers has increased. Better tools are now available to analyze code for potential issues. And whereas games were originally distributed on ROMs or systems without network access now fixes can be pushed to consumers in minutes. So much so that newer games sometimes make balance adjustments on a regular basis using analytics feedback. [8]

However, quality assurance continues to be a real problem. RAGE was developed using the aforementioned static analysis tools extensively [9], with industry veteran John Carmack working as technical director. When the game was released, incompatibility with many themselves complex graphics drivers meant a large proportion of customers had a gaming experience that was not satisfactory. Carmack himself in the following year's keynote apologized for "...the release of the pc [game] not working on almost half of our customers'...". [10] Due to bugs only present when the game was paired with specific graphics drivers it took many months before the game was widely playable.

For some smaller developers, the issue is further compounded by the licensing of engines or tools from companies as they are building a game with software they did not themselves make which might have design facets they do not understand. As the source code for said engines is

rarely available [11], it is difficult for these developers to pin down bugs precisely and impossible for them to fix them beyond reporting them and hoping the company who made the tool fixes the bugs.

2.1 - Code Reuse

To write reusable code is to write code in such a way that it has few interdependencies with other components and can be used with other projects cleanly and efficiently. A library, in itself, is an example of reusable code in many cases. However, it is common for a more complex codebase to include many similar implementations of the same ideas and technology. For instance, AI might want to be able to determine if it has line of sight with another object. A gun might need to be able to tell what it hit. A light source might need to be able to tell if it is occluded. These are all basically the same thing; a "raytrace", which objects a line between two points intersects, if any. With prior knowledge of what each part of the engine might need to do, this can be planned from the start. However, different parts of a game engine are typically developed by separate people or even separate departments now, and without exceptional management helping the programmers to work together, they might not even realize how much workload they could share if the functionality was made common.

2.2 - Platform Constraints

Adding to the troubles of low-budget development is that the platforms generally favoured for their low cost of entry typically have low hardware capabilities, [12] and the research and development necessary to produce a heavily optimized engine is expensive and often incurs the very complexity of code that makes it hard to debug and test.

A performance limitation specific to mobile devices in particular is draw calls. [13] Regardless of the content of an instruction to draw, each graphics operation has an overhead in which the drivers decide what to do, and then tell the GPU to do it. Due to the asynchronicity of graphics pipelines, [14] [15] this overhead is difficult to distinguish from the content of the draw call without vendor-specific performance analytics tools. The recommended limit under Unity for the iPhone 3G is 20 per frame, [16] though on newer hardware this is far higher. [17] The limit is defined more by how the components of the device are put together than the components themselves as it is communication between components that forms the overhead. [18] Draw calls which require state changes such as switching texture or changing blend modes require further overhead.

Fillrate is the number of pixels the GPU can populate in a given time span. The iPhone 3G's SGX GPU, not as fast but similar to the Adreno 205 in the Lumia 710, has a quoted optimal fillrate of 100 megapixels per second. [19] The Lumia's 800x480 display [20] contains 384,000 pixels, which will be refreshed 60 times a second. In optimal circumstances, this means the screen could be fully refreshed 4.34028 times per second, however, this optimal figure was likely created under specifically crafted conditions (no lighting, no textures, etc.) and is not

representative of the final results; it is only for comparison with GPUs of a similar architecture. If multiple, non-backfacing triangles stack up (such as three walls in front of one another) each has to be filled, and then overwritten when the next triangle is drawn, thus the fillrate is still impacted by each. If the new pixels are behind what was already there, they are culled by the depth buffer test, but there is still some fillrate cost. Additionally, transparent geometry is especially expensive, as it does not write to the depth buffer, and involves blending mathematics. Due to limited memory bandwidth, geometry with special techniques like dual blended textures are extremely intensive on fillrate.

Modern GPUs are simplified in that they can only draw triangles, and the SGX datasheet provides a best-case-scenario of 1 million triangles per second, [19] which at 60 frames per second, is 16,666 triangles per frame. Again, however, this is an "optimal" figure, and not indicative of real world usage. The cost of geometry on today's hardware is in vertex processing as opposed to triangle count. A basic vertex must be run through the 4x4 transform matrix, which in itself is 12 floating point multiplications and 4 floating point additions, before perspective division, three floating point divides. If there is a lighting normal, this must be run through a 3x3 rotation matrix, which is nine floating point multiplications. This is the kind of massively parallel repetitive operation GPUs are good at, but the figures do add up quickly. In total, this is 28 floating point operations per vertex. Though identical vertices are only processed once (as a result of indexing; the vertices are specified in a list, and then a list of vertex IDs is used to form a sequence of triangles) if any data in the vertex is different, it counts as a completely different vertex. Though the location data may be a 3D vector, the normal may be a 3D vector, and the UV coordinate a 2D vector, the vertex is taken as a single 9D object. This means any normal or UV splits also increase the geometric cost of an object. [21] Some recommend comparing the geometric cost of objects using the final vertex count as opposed to the polygon count, as it is more relevant in a modern context. Similarly, as model vertex densities increase, per pixel lighting is sometimes cheaper than gouraud shading as the object covers fewer unculled pixels than there are vertices. Figures vary on recommended total scene triangle counts, but generally this is around 30,000 triangles. [22]

2.3 - Combat Pool

The example game being built on this research to test the hypotheses proposed is a fairly simplistic but technically broad game called Combat Pool. In this, players control a robotic pool ball on a team to try and pot the other team by firing at them, knocking them into pockets. This game will allow the demonstration of a semi-dynamic lighting system, artificial intelligence, solid physics and how all of the elements come together to build a finished demonstration product.

2.3.1 - Asset Budgeting

Though the game is small in scope, it is still important to consider system limitations. Memory is not much of a concern on the target platform; the maximum amount of memory allowed for the current app on Windows Phone 7.1 with 256Mb of RAM is 90Mb. [23] Most of the game content is reused heavily in the same scene. The screen is not large, so textures do not need to be very high resolution. Phones typically do not have dedicated VRAM, sharing it with system

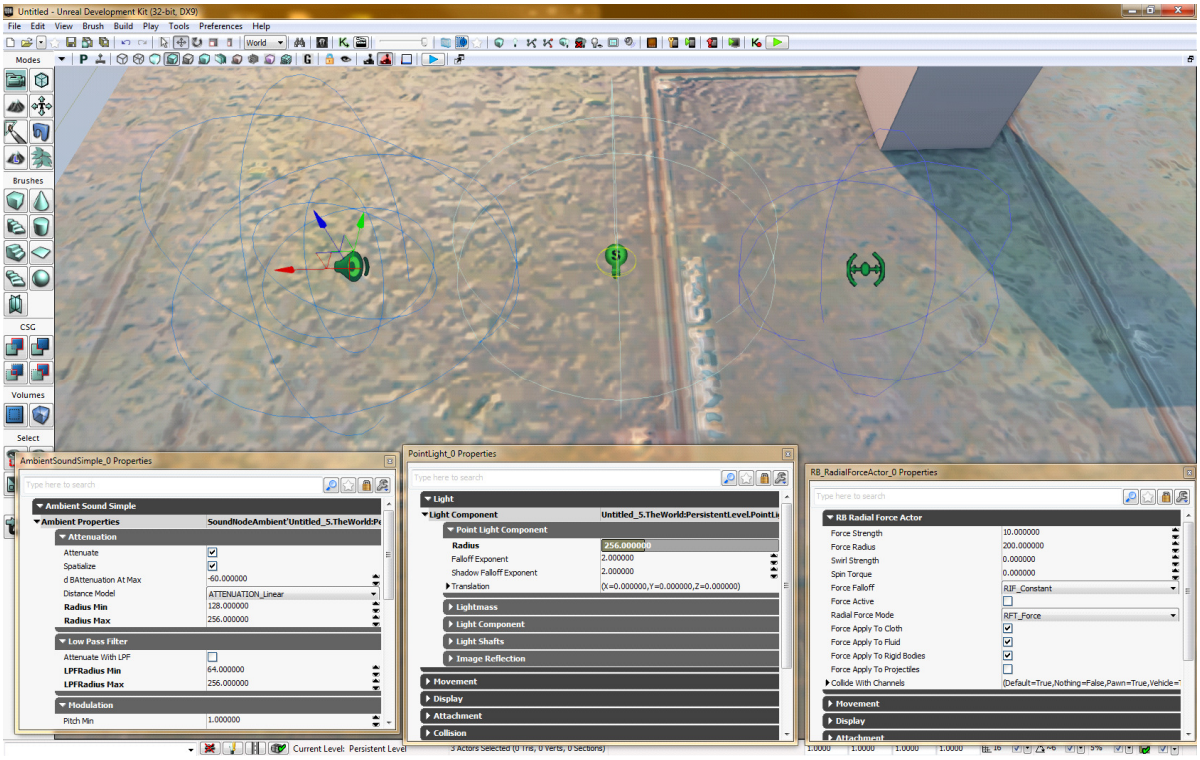
RAM. Geometry is not expensive memory wise either; 1000 vertices with UV coordinates and lighting normals come to just 31.25kb using single precision floats. Much of the cost is in the processing and display of content, and possibly in load time.

Each team has up to 7 players, with one extra player, coming to 15 players in total. If each player model is about 1500 triangles, this comes to an absolute maximum of 22500 triangles for players without LOD. If the game begins to slow down with more characters, this leaves the choice of reducing the number of players (and it remains to be seen if the number of players is even appropriate for gameplay) or producing lower quality models for other players, to be used either at a distance or for all other players. This leaves about 10000 triangles for the environment, which seems appropriate given the simplicity of it.

3 – Research

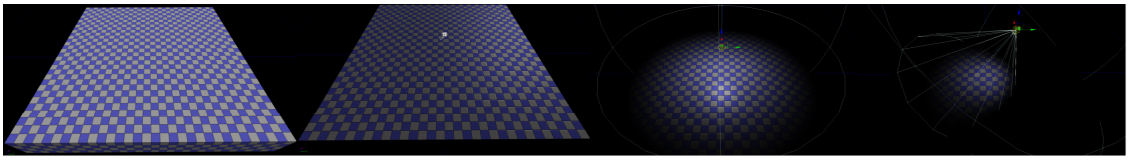
Now, to investigate the features often available in modern games engines.

3.1 – Falloff



[24]

A common concept in games engines is falloff; a value that is variable dependent upon where it is sampled from. These are usually cached in an optimization structure of some kind to quickly look up which influences an object should take on and which can be rejected early. Many engines also make a distinction between dynamic and static falloffs, computing quick look up tables for static falloffs such as lights. [25]

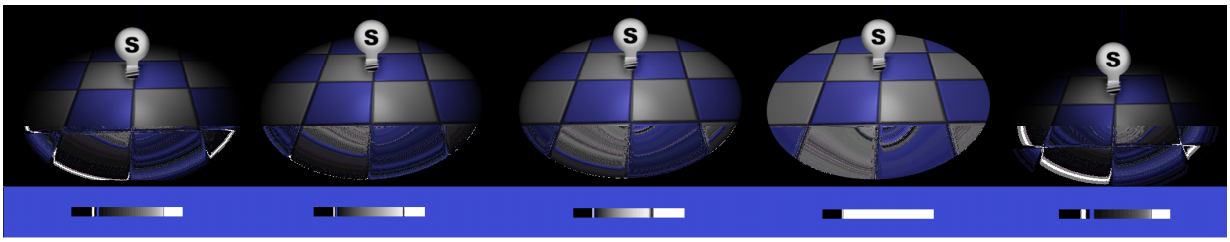


Skylight Directional Light Point Light Spot Light

[26]

Many systems with falloff provide a set of shapes that can be used. The most common is the sphere or point, which has a spherical falloff from a central point, with a flexible radius. Some offer a "shelled" variant with a second, smaller radius. The intensity inside this smaller sphere is a constant 100% before interpolating down to 0% at the edge of the larger sphere as per normal. Cones are also common, implemented as a sphere with an orientation. The dot product between this orientation and the normalized vector between the centre and the sample point is multiplied against the spherical falloff, so a "cone" of intensity is created.

Ambient or directional falloffs also exist, but their implementation is sometimes as a "global" setting of the game world. Examples might be gravity, ambient lighting, or ambient sound. Ambient falloff is at a fixed intensity no matter where it is sampled, while directional falloff is the same but coming from a specific direction.



x^2 x^1 x^0.5 x^0 x^4

[27]

Finally, linearity of the falloff is sometimes variable. This offered either by a dropdown providing a selection of standard falloff curves (linear, quadratic, etc.) or recently an exponent value applied in normalized space. This allows for a variable rolloff which can be useful for simulating different types of light.

3.1.1 - Light

Light is one of the most important and generally expensive elements of a renderer. Without good lighting, a 3D scene tends to look more like a collage [28] than a believable world. In a games engine, lighting is often divided into static and dynamic sources, with multiple "shapes" used to simulate real-world light sources. Colour may also be set. [29]

The cost of lighting a 3D world in real-time comes from having to perform the following on

every single lit surface in a hypothetical scene lit by point lights without radiosity, specularity or shadows: [30p.253-255]

- Build a surface normal
- Find the difference between the pixel's world space location and each light source's origin
- Measure the difference to find the falloff intensity of each light
- Normalize the difference and dot product with the surface normal to produce a second falloff intensity
- Multiply the two intensities together and by the colour/brightness of the light, take the sum of all the lights' results and multiply by the surface colour.

This makes doing full-scene per-pixel lighting extremely heavy on fillrate, and as GPUs are far more efficient executing "simple", branch/loop-free code than they are iterating through a list of lights, the implementation is often more complex. Forward renderers, for instance, draw the object once for every light, using additive passes that are very expensive, while deferred renderers draw the scene to VRAM-heavy buffers and draw lights as 2D passes over the top.

None of this is practical on mobile devices, as neither technique applies well to the limited, relatively slow VRAM often shared with the CPU. The very first 3D games used specific art assets with lighting painted straight into assets' textures, [31] which was time consuming, limited the versatility of the assets, and not entirely believable as there were no dynamic factors to the lighting such as specularity or real shadows.

Some games moved to using interpolated vertex lighting. [32] This involved calculating a light colour at every vertex in the scene and linearly interpolating to find the colour of the fragments. This is indeed cheaper, but has many drawbacks. Chiefly, without modelling specifically for this technique by tessellating surfaces and avoiding long edges, lights may not show up on large surfaces or cause very obvious streaks as the vertices spuriously intersect with light sources.

Others used light mapping [25] for static surfaces; a second UV channel shared between static assets throughout the scene precomputed before play. This technique contributed to the improved graphics of Quake, though the technique vastly increased the workflow length as building lighting on machines of the time could take days.

This, however, left dynamic objects in the scene without lighting. Some renderers opted for gouraud shading [33] (which in the context of a relatively densely tessellated character model did not have as many problems as applying it full-scene) while others sampled a single point in space, sometimes from the nearest lightmap texel, and lit the character based on that value.

Mobile platforms have access to reprogrammable GPUs [34] and more processing power; so

many modern games appear to use a combination of lightmapping for static objects and normal mapped per-pixel lighting for dynamic objects. Many console games use complex mathematical solutions to apply normal maps to lightmapped objects, but mobile devices have neither the power nor the memory to do this well yet.

3.1.2 - Audio

Games have not advanced as far in the field of audio as they have graphics since the playback of recorded audio became feasible. Increased CPU speed and memory have enabled more varied, higher quality samples, (sometimes with real-time decompression or effects such as reverb) and games tend to feature more ambience than they did before, but from an engine development and level design point of view the field is nearly identical.

Sources are imported and placed in the scene, as either full-scene ambient or spherical radius sources. Volume and playback rate can sometimes be modulated to produce effects, and full scene effects such as reverb or filters can be applied to enhance the impression of a change of environment. (Such as being underwater or in a large echoing hall) [35]

The spatial effects of audio are well researched, documented and implemented in modern games engines, producing a "3D" feeling audio space in which players can locate a sound source without being able to see it using stereo effects and processing to make sounds appear to be coming from behind.

As audio sources do not require any scene-specific processing that can be precomputed, engines do not discern between dynamic and static sources.

3.1.3 - Physics Attractors

Often implemented directly in the physics engine, attractors can be added to a scene to apply forces to physics objects. There are many different types, but generally these are divided into persistent and burst attractors.

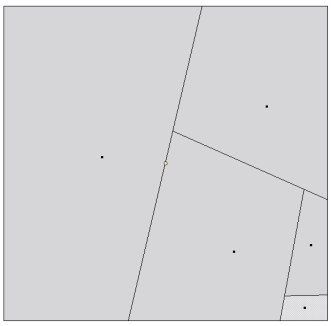
Persistent attractors continuously apply a force, such as a black hole, air being sucked out of a spacecraft or the jet wash of an aircraft. Typically these can be placed in a scene during level design and turned on or off using scripts. Gravity is often a special case despite being functionally similar to a directional physics attractor.

Burst attractors are used for instantaneous forces such as explosions and hit impacts. Internally, these would not be considered an object; there is little point in allocating memory for an object that will only be in the scene for a single frame. For level design purposes, a marker may be placed in the scene that produces a burst when triggered by script, but generally these are used more by objects themselves such as projectiles.

3.2 - Optimization Structures

Critical to performance in a real-time environment, especially a large scene, is the optimization of space. If the scene can be broken down into zones and entities an operation interacts with shortlisted to the zones the operation intersects, this drastically accelerates the speed of the operation.

3.2.1 - BSP



Binary Space Partitioning [36] entered the mainstream games industry with Doom, where it was used to drastically accelerate rendering of its world by splitting it into convex sectors with precalculated Potential Visibility Sets. In BSP, the world is cut into a tree of sectors using a single plane per sector to cut it into two smaller sectors. This makes searching a BSP tree to find the leaf sector that contains a point a very simple and fast operation, as checking which side of a plane a point lays is a very fast dot product test. Each sector is also convex which may accelerate other operations.

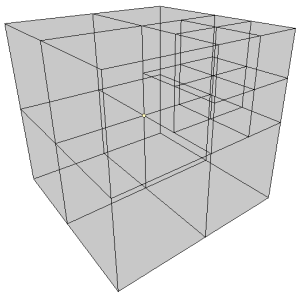
The problems with BSP, especially on newer platforms, are four-fold. Doom used the same geometry for graphics, physics and AI. It greatly accelerated rendering because scenes were very low polygon and every triangle was very expensive. Now, however, individual triangles are far less expensive than fillrate and batches. It is not possible for a game to tell the graphics card which triangles in a batch to omit, nor would it be economical to do so as so much data would need to be uploaded every frame, straining the already premium CPU-GPU bandwidth.

Secondly, the acceleration structure would be so complex for a modern scene, that generating it, storing it and reading it would all be so heavy on memory in particular that it would be to the detriment of performance.

As most BSP trees are generated algorithmically, it is up to the implementer to ensure that the tree is optimal for its intended use. Often, the topology of BSP created using Constructive Solid Geometry is uneven and sub-optimal, [37] and no artist/level designer tools exist to allow the modification of how it was generated. That BSP implementation is usually specific to the engine means no generic BSP editor tools like 3DS Max to polygons exists. Instead, editor tools create a list of additive and/or subtractive objects which are "compiled" into a finished BSP. [38]

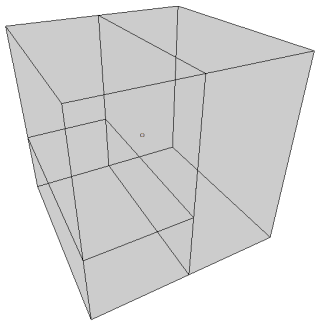
Finally, as most modern games don't use the same geometry for drawing as they do physics or AI, a separate acceleration structure would be required for each, further compounding the memory overhead.

3.2.2 - Octree



Derived from the 2D quadtree, an octree defines 3D space as a cube, which subdivides into eight equal cubes half the size when a condition (such as more than a specific number of objects in a single cube) is met. This structure is very tuneable, can vary its density as required and is very predictable compared to BSP. However, the structure is less flexible geometrically, meaning to represent a surface such as a slope properly many subdivisions are necessary.

3.2.3 - KD-Tree



Similar to octrees and quadtrees, KD-trees [39] treat the entirety of 3D space of a cube, but instead of subdividing into equal cubes KD trees split into two cuboids by creating a plane along an axis. This approach allows the engine to break down scenes optimally, is faster to search than standard BSP (single floating point compare instead of dot product) and is tuneable like an octree.

3.2.4 - Raycasting

Many of the standard spatial optimization structures used today were designed for use with software raytracing renderers, and so are a good fit for a generalized raycasting algorithm to be used by physics, shadows, AI line of sight or hit tracing. They are well suited to this purpose, but there may be better solutions for use in games engines.

3.3 - Artificial Intelligence

There are many components of a solid, believable AI system that must come together to create a challenging, interesting opponent. First of all is pathfinding, in which given a goal point an AI agent must be able to determine a route which satisfies their physical constraints. (Size with

holes, jump distance with gaps, etc.) Although in some cases such as games using snap together complete rooms there is no need to have pathing information specific to the level, most engines use a solution where pathing information is built specific to the level rather than any assets.

Other aspects, such as line-of-sight checking, would be handled by systems already discussed, while some, such as state machines, are very particular to the game in question and will not be examined.

3.3.1 - Standard Optimization Structures

Games have in the past used their world's structure directly to operate artificial intelligence. Examples such as Doom to Quake 3 performed pathfinding using precomputed data built using the BSP, but as BSP fell out of favour so did using this technique. Many 2D games based on a tiled grid for graphics have used that same grid for collision and AI pathfinding too.

The problems are that the optimization structures in use in a game define the shape of the world, and possibly which areas have line of sight to others, but not which are accessible. The key difference is that a ray has a thickness of zero, able to pass through any gap, and is not subject to gravity unlike most characters in games. The structures also do not define how to access a location by navigating around objects, typically only whether they can be accessed in a straight line.

3.3.2 - Waypoints

A common solution is to allow level designers to place waypoints [40] in the level, which at build time are quite expensively processed, determining if there are any gaps to cross, how far an agent would need to jump to reach a higher waypoint, and generating a look-up-table of which is the quickest path to a specific waypoint from each one. In the game, an agent then merely has to find the nearest waypoint and follow the directions, which is very fast.

However, this defines the world to AI as a network of infinitely thin corridors throughout the level, which produces unrealistic behaviour. Large open spaces are often filled by a single point in the centre with an array of linkages running to the entrances and exits, which results in all AI entering and exiting the room running to the centre, then running to their chosen exit even though this might quadruple the time taken and cause traffic congestion due to many agents trying to pass through the same point. AI always takes the same route around a corner or down a corridor, making them predictable.

Waypoints can have minor problems with moving geometry such as lifts or for instance collapsing platforms, often requiring special case waypoints which can be turned on or off. This causes problems with precomputation, as the look-up-table may be invalidated by changes to the level.

3.3.3 - Navmesh

An expansion on waypoints is a navmesh, [40] which is an invisible, low polygon model running throughout the map. Anywhere on the surface of this model is defined as a valid place for an agent to be, and like waypoints these can be precomputed for very efficient navigation of a largely static world.

The key advantage is that rather than defining a network of paths, a navmesh defines a network

of spaces agents can explore. [41] This means agents can make their own paths through open space, walk around dynamic objects that may have fallen on the navmesh (or other agents) and make erratic movements such as dodging weapons fire without leaving their view of the world.

In level editors, navmesh tools are either automatic such as the Pylon system [42] in the Unreal engine, or involve a system similar to retopology such as the Fallout 3 navmesh tool [43], in which level designers click in the viewport to draw polygons. As the navmesh is not seen or collidable and only roughly defines the space, it does not need to be precise.

However, other problems inherent to waypoints persist; disabling or enabling areas invalidates precomputed routing data, complicating the implementation so that doors or lifts can be made functional.

Common to all of these pathfinding strategies is that they do not account for dynamic objects that may have been placed, such as a car crashed over a road. In reality, the driver would just turn around and find another route, but this is an extra complication.

4 - Design

4.1 - Falloff

The Unreal engine presents completely different interfaces, [24] presumably built on completely different code, for performing similar tasks when dealing with entities with a falloff. In the proposed codebase, all entities using a falloff such as lights, sounds, and physics attractors have been refactored using a generic falloff interface. This interface allows for sampling of an object with falloff from a point in various ways including intersection testing, intensity and direction. (For use with lights or physics attractors, which need a normal) This should decrease functionally duplicate code.

4.1.1 - Shapes

The standard falloff shapes such as spherical, conical, ambient and directional ambient were designed for use with the falloff system. Spherical will be a falloff to 0 at a specific radius from the origin, with the normal pointing toward the origin. Conical is an extension of spherical with a spot normal and radius multiplied calculated using a dot product and multiplied by the spherical falloff. Ambient is an omnipresent intensity that is always intersecting, while directional extends ambient to include a specific normal.

4.2 - Navmesh

Core to the at this stage hypothetical engine is the navmesh. Expanding upon the typical idea of a navmesh being used to define space for the AI, players will be confined by the navmesh's surface. It should be much easier to write physics code that works in a semi-2D domain on the surface of the navmesh, without having to write code for collision with scene objects, just the borders of the navmesh. If the players cannot escape the navmesh, this simplifies writing AI code as no glue code is required to determine where the objects are in the navmesh after the physics engine moves players. Finally, if all dynamic objects are confined to this space, objects with falloff can be cached in the tiles of the navmesh. This overall removes the requirement for separate physics, AI and optimization structure code, vastly reducing the complexity of the finished product if requiring extra time spent designing the system before implementing it.

5 - Implementation

5.1 - Scene Manager

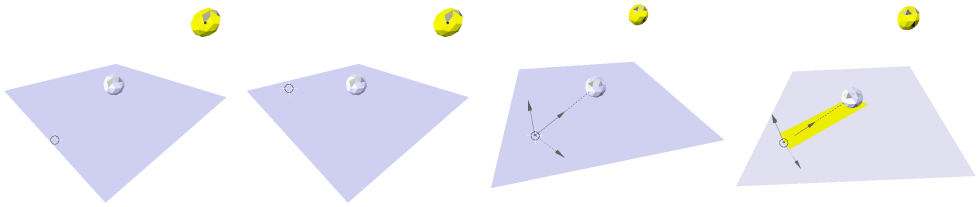
The beginning of the game engine was the scene manager, which allows a scene to be imported from a file rather than generating it with code manually. This is not deeply important to the hypothesis, but forms a necessary basis. A secondary, separate class structure exists designed for easy serialization which references the "true" real-time class structure.

5.2 - Falloff

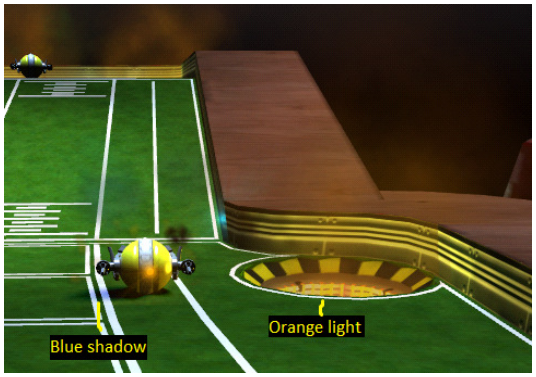
5.2.1 - Lighting

Dynamic objects with lighting iterate through the list of cached falloffs in their current navmesh tile, and those that are lights are sampled. XNA's BasicEffect exposes three directional lights with both diffuse and specular [44p.243-245] and optional per-pixel lighting. [44p.238] while these channels could be heuristically assigned, the channel is explicitly set in the light to prevent numerous close, small, bright light sources consuming the three channels, ignoring general scene lighting. If multiple lights exist in the same channel, a weighted average is performed by intensity on normal and colour. The result is that when multiple lights contest for a channel, they merge into a "blob" of light, which is not an undesirable result compared to sudden switching from one to the other when the threshold is crossed.

As important as light is shadow. Good shadowing enhances depth perception by showing how distant an object is from a surface. Although on Windows Phone 7 there is not sufficient power or flexibility to build a "true" shadowing system using a technique such as raytracing or depth buffer shadowing, the effect can be approximated. It was common in older games to draw a shadow by drawing a dark transparent plane below dynamic objects featuring a blurred circle, but using the lighting information this technique can be expanded upon. By using a plane for each channel and orienting it away from the light source, a shadow semi-realistically tracks as the player passes a light source.



This is done by taking the difference between the location of the entity and inverting it to get a point behind the entity when looking at it from the light. This is then projected onto the navmesh triangle, producing a line along the navmesh. Taking the normal of this line, and cross producting it with the normal of the navmesh triangle produces a side normal. Multiplying these normals with the radius of the entity and adding to the start/end of the line produces a rough approximation of the shape of a shadow.



A trick that helped improve the perceived lighting contrast was to use a custom blending mode to remove the colour of the light source from the ground. This means a blue light casts a yellow shadow. This would not look right in some scenes, but in typical high hue contrast game worlds looks somewhat better than desaturating modulation.

This does not however work well for overhead lights as the shadows are short and do not show up well. A fourth plane was added for a form of ambient occlusion, which helps visually cement the object on the surface better.

5.2.2 - Audio

Unlike light, audio is not sampled from the player object, but from the player camera. This means the audio cannot be cached in the navmesh (as the camera is not restricted to the navmesh) so audio sources are kept in a list global to the scene. Some objects in the scene use several samples at differing pitches/volumes but attached to the same falloff, so this was optimized for by grouping sounds under a container with an attached falloff.

To prepare for a multiplayer environment, support for multiple audio viewports [44p.92-96] was implemented from the start. Using a similar system to lighting, each viewport iterates through the list of audio sources and calculates a volume level and panning value. After this, the results are averaged, weighted by distance.

This was done instead of using XNA's own spatial system (which is more advanced in many ways and is likely hardware accelerated) because of inherent limitations in XNA's audio system. There is a global limit on the number of concurrent playing sounds, as well as a significant processor

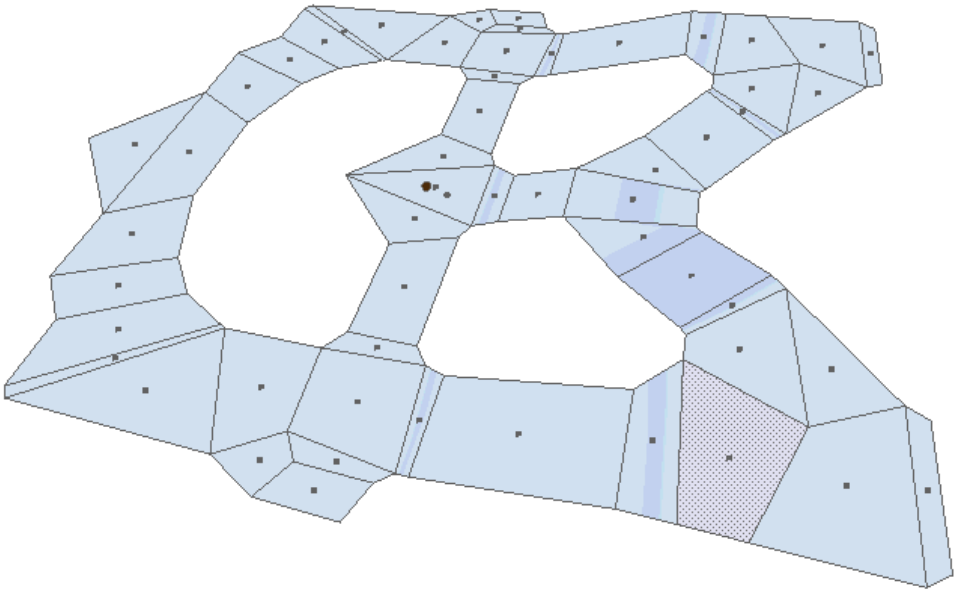
time impact on mobile devices in particular incurred by software audio mixing. This is made worse by how XNA handles 3D audio; rather than using a radius with a linear falloff to zero, it uses a realistic inverse square falloff which never fully depletes. [45] Combined with multiple simultaneous samples per character, this instantly flooded the concurrent sound list and caused slowdown. Managing the audio manually with a falloff to zero mitigated the problem.

5.2.3 - Physics Attractors

Every frame, physics actors also iterate through their parent navmesh tile's list of cached falloffs to find attractors. The physics actor's velocity is incremented by the intensity of the falloff along its normal, thus repelling the physics actor.

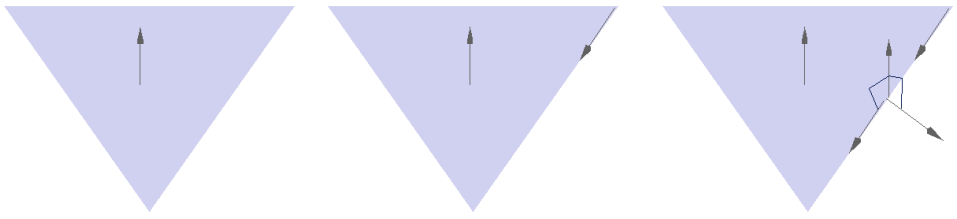
Gravity was implemented using a directional physics attractor, while the pockets spherical physics attractors using a negative intensity to attract rather than repel physics actors.

5.3 - Navmesh



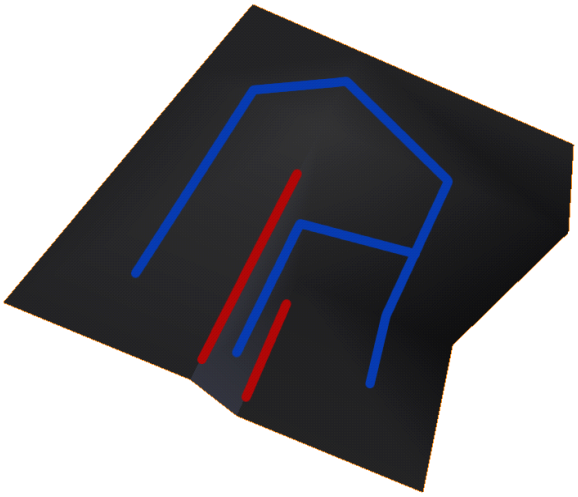
Creation of the navmesh was integrated into the workflow of producing a level for the game by using it to prototype the flow of the level before any visual elements were produced. The prototype asset was modelled in Blender.

5.3.1 - Import



The game loads in a single model listed in the scene's header, allowing XNA to handle the import and interpretation of the model. These are converted into a list of navmesh tiles by iterating

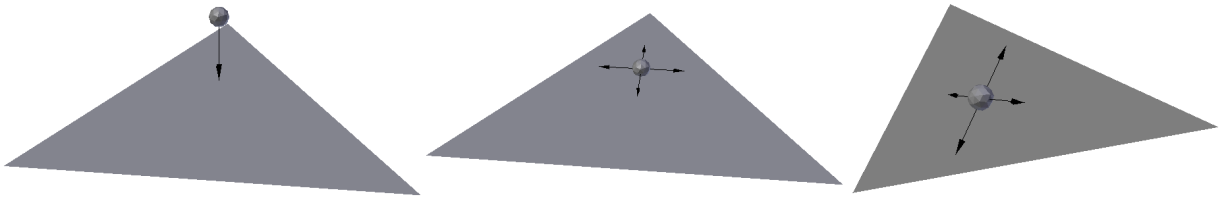
through the properties of the Model. On generation, a navmesh tile also builds a surface normal, a normal along each edge and a cross product between the surface normal and each edge normal, used for detecting when entities leave over a border.



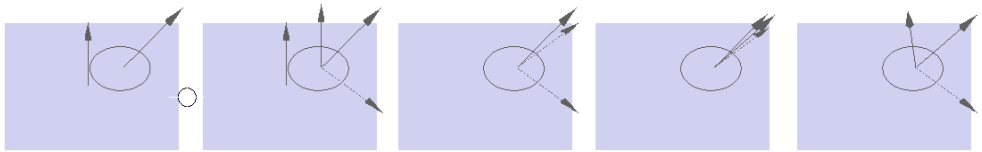
Afterward, the parent navmesh adds references between the navmesh tiles so tiles can find their neighbouring tiles. This is built from vertex indices so that splits can be introduced using smoothing groups if need be. [30p.215-192]

5.3.2 - Physics

5.3.2.1 - Surface Crawling



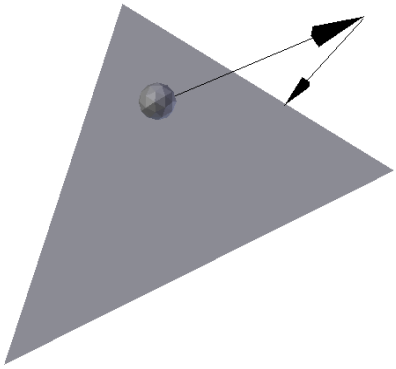
Physics actors move along the surface of the navmesh tiles. A method was added to the navmesh tile class which projects a world location onto the navmesh tile. This is used to make them stick to the surface. Velocity is similarly nullified along the surface normal.



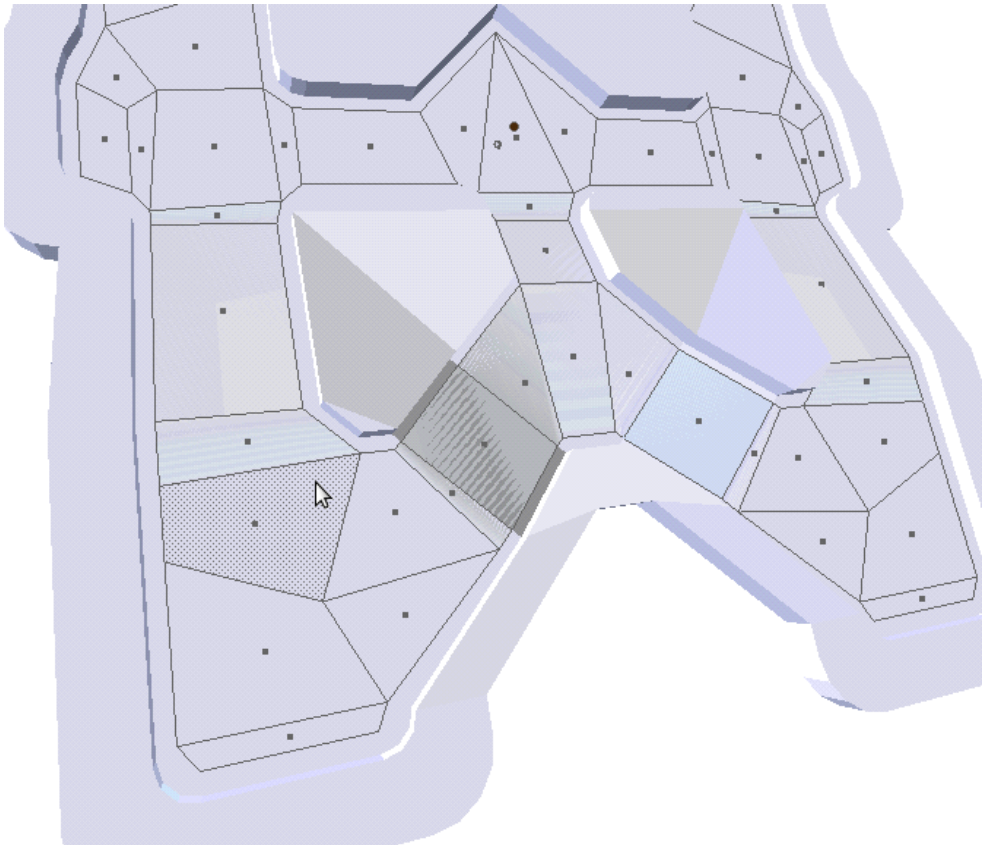
Rotation is interpolated to match up with the surface. The physics actor's forward vector is cross producted with the surface normal to create a vector pointing sideward. By cross producting this with the surface normal, the "correct" forward vector relative to the surface is calculated. By interpolating between the original forward vector and this, and performing a cross product between the result and the side vector, an interpolated up vector is produced. These three axes are used to build a new rotation matrix. This negates roll relative to the surface for the time being. [46p203-214]

5.3.2.2 - Collision

When the origin of the physics actor exits a navmesh triangle, this is detected by the dot product between the difference between the actor and a vertex along the edge against the edge's side normal inverting. This only needs to be done for the three edges in the current triangle, making this a very cheap procedure regardless of the size of the navmesh.



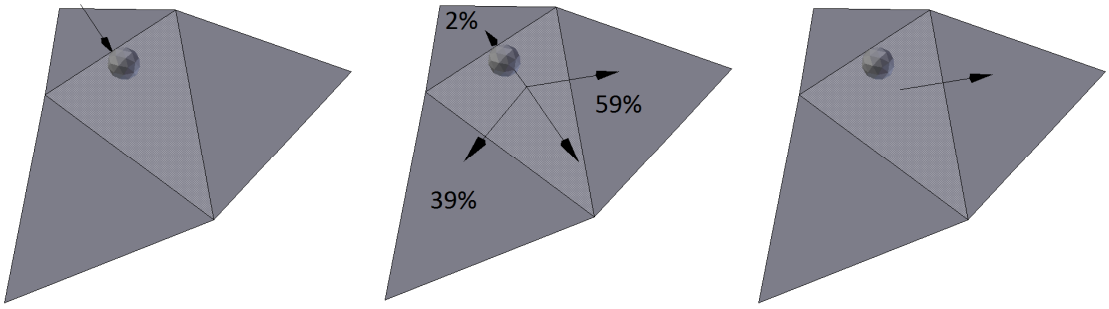
If the neighbour reference along that edge is null, the actor's location is projected onto the edge, producing sliding collision. If the result moves into another triangle, the process continues until the sequence ends.



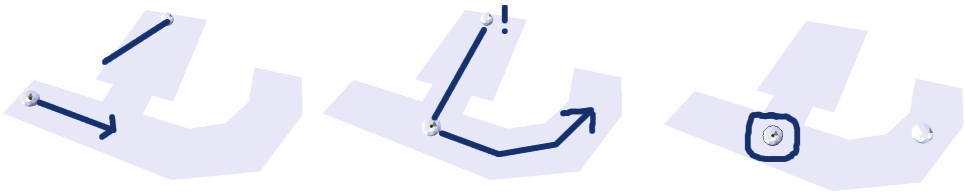
This, of course, treats all physics objects as being an infinitely small point. Adding support for arbitrarily sized objects is possible but would greatly complicate the physics code for little benefit in the context of the demonstration project. Instead, when producing the visible model a border was added the same thickness as the radius of a ball.

5.3.3 - Navigation

Player inputs were abstracted out into a controller interface so artificial intelligence and network players can operate the same vehicles. This is further abstracted internally so that the artificial intelligence generates a target location through its state logic. By taking the difference from the vehicle's origin and performing a dot product against the right/forward axes of the transformation matrix, it can be determined whether the target is left or right, forward or behind, and generate input for the controls accordingly.

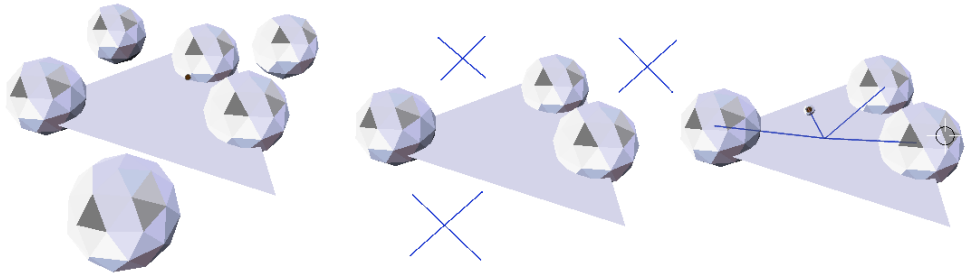


In the default state, the artificial intelligence wanders the navmesh at random. On entering a navmesh tile, the AI examines the edges with neighbours, picking an edge by random weighted by how well it keeps going in the same direction. The centre of the neighbour over that edge is then set as the target.



An AI agent spots enemies by performing a dot product to determine if it is in the field of view, and then performing a raycast along the navmesh. This, of course, is not entirely realistic; not taking into account holes in visible geometry, but allows the player more information than them which makes it more fairly balanced. If line of sight with the target is lost, their previous known position is retained, only reverting to wandering mode when the AI comes within a specific range of it. As the AI can move to any point on the navmesh, and will only attempt to move towards points it has line of sight with, this should prevent the AI ever getting stuck in thin sections of navmesh; if there is no suitable path, it will not get line of sight.

5.3.4 - Caching

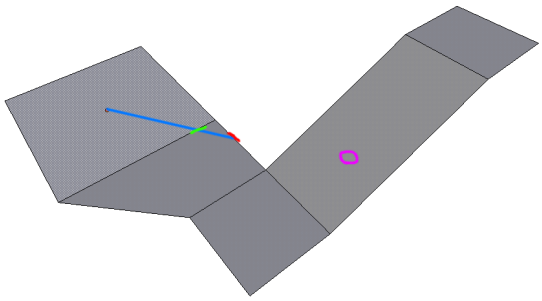


Caching of entities with falloff in the navmesh is done through an interface allowing an entity to expose a falloff. The navmesh itself then performs an intersection test against each triangle, adding a reference to the entity to each triangle it intersects. A potential problem is that the list

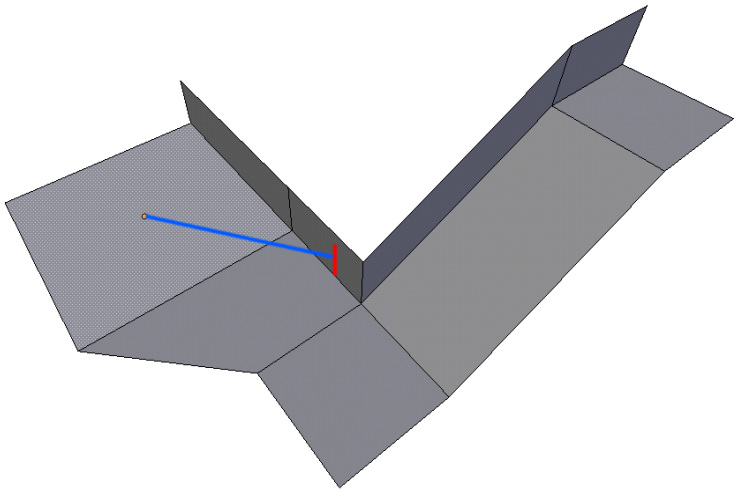
is of all entities which have an intersecting falloff, so a search for lights also lists physics attractors and audio sources, even though they are not useful, wasting CPU time discarding these through a type check. It also requires a type cast to retrieve light specific properties. This could be partially mitigated by using a dictionary with entities also returning a "category" string or type to be split into separate collections by. [47p.129]

5.3.5 - Raycasting

Navmesh raycasting works by taking the start and end point and determining which tile they are in. The end point is projected onto the plane of the current tile. If it is in the current tile but the current tile is not the end tile, it assumed to be unreachable as it's likely on a different level vertically. If the point is over an edge without a neighbouring navmesh tile, the cast hit a wall and stops. Conversely if the edge does have a neighbour, the ray is projected onto the edge and the search begins again from the next tile.



This worked well in flat environments, and as it only had to factor in the tiles the ray directly intersected had very good performance. When faced with hilly terrain, the trace would completely ignore the variations in altitude. An alternative, more standard raytracing algorithm was also written that worked on the visible models in the scene. Due to the lack of a strong optimization structure for this, and that most of the scene models were merged to improve draw call performance, this was quite expensive. An additional runtime cost was incurred too as the GetData() method of XNA's Model class appeared to stall the graphics pipeline on mobile devices, necessitating a cache of that data.



These two methods of raycasting the scene geometry were used in tandem in the end; the navmesh method was used to detect what the player might be aiming at, and the visual method was used to see if that target was a clear shot.

5.4 - Particles

Though the game demo required a number of different particle effects, all were "simple" particle effects easily represented by a billboard and a basic set of rules for motion. It was observed that XNA's SpriteBatch [44p.62-79], though likely not intended for this purpose, handles batching of geometry generated CPU-side. SpriteBatch is likely to have lower-level access and "know better" how to generate and upload that data in the most efficient way possible for the hardware available. It can also optionally Z-sort the sprites, which while CPU intensive with large numbers of particles, with the amount in use, proved practical. To make SpriteBatch work in 3D, all that had to be done was to transform the world space locations into screen space (not normalized device coordinates; SpriteBatch works in pixel coordinates, likely reversing this in the backend as it is designed for drawing 2D sprites) and calculate the appropriate scale. To allow different particles to be part of the same batch, all particles are taken from an atlas texture. All particles also use the same blend mode, premultiplied alpha; to get both additive blend and alpha blend, additive blend surfaces were hand tweaked after premultiplication to contain the original RGB values and an A of zero.

5.5 - Optimizations

The Windows Phone 7 profiling tools help to identify methods that consume the most CPU time, but as the main game loop runs outside of the codebase (in XNA, developers override the Draw() and Update() methods in their Game class derivative) it can be difficult to gauge the performance of the graphics pipeline in particular. Graphics APIs are highly asynchronous by design, and while a call to upload a texture or draw a batch may return immediately, the instruction has merely been loaded into a queue and it can only be certain the operation actually completed when the results are requested. (Either by asking for a buffer swap or sampling a pixel colour) This made determining performance roadblocks a process of elimination combined with guesswork. [14][15]

A surprising source of performance loss was the inclusion of a textured skydome. Despite being low poly, with a relatively small texture, and no fillrate-intensive properties such as lighting, whenever a large portion of the screen was taken up by the skydome framerate would drop off dramatically. As there was not much detail, this was replaced with a vertex painted skydome, which restored performance.

The order in which objects were drawn proved important. Performance shortfalls were frequent whenever a number of enemies were close to the screen. This was to be expected due to the high fillrate cost of the phong shaded character models. Switching down to gouraud shading at a set distance would not help; at a distance at which it would not negatively impact the quality of the image, the vertex cost incurred by gouraud shading would likely outweigh the fragment cost of phong shading. [30p.213-214] The shadows, with their custom blend mode, were at this stage being drawn before the character models. This meant that the reasonably expensive shadows were being drawn first (and without z-write, so all four layers' overdraw cost stacked up very quickly) and then wiped away by the solid characters drawn on top. Moving shadows into their own pass after solid objects and before blended objects mitigated this problem, and likely caused other performance improvements; by grouping similar operations, context switches will have been reduced dramatically.

It was noted that an occasional hang was occurring on hardware. Profiling showed that on occasion, CPU time would remain close to, or at, 100%, but little to none of it would be accounted for in the method breakdown. Using the memory profiler showed that in each one of these instances, the garbage collector had stepped in. The VM available on Windows Phone 7 at

the time used a "Stop-The-World" garbage collector, triggered when memory usage went over a specific limit. This freezes all threads in the application, then checks for objects that are not in use, and then resumes the application. During this, no events or messages are responded to, and the program may appear to crash. To prevent this, especially in a real-time context, reference objects should not be instantiated on a routine basis. Memory profiling showed that the vast majority of the discarded memory was particles, so by limiting the number of particles in the scene, generating a pool of them that size and using free particles rather than allocating new ones the issue was averted.

5.6 - User Interface

A simplistic user interface was required for the main menu and pause menu, as well as a title screen. Originally, Silverlight was used for this purpose, but there is not an implementation that can be reused cleanly outside Windows Phone. Additionally, performance was poor when the Silverlight overlay [48] was in use. An experiment from a separate project was brought in, the Presentation layer. This used SpriteBatch to draw a hierarchical GUI (not dissimilar to GTK [49] in that controls divide up a finite space) compatible with a mouse or touch screen, with scope for gamepad interaction. Additionally, a transitions system using render-to-texture [50] was used. By storing the previously viewed scene/menu in a texture, this can be drawn on a plane and manipulated easily for cheap fade/slide/zoom transitions.

6 - Evaluation

6.1 - Effectiveness

The techniques outlined herein proved effective in making a game demo quickly as a single developer also making the content, reducing the time spent on code and assets. Production took twelve weeks before stopping, but in that time a near-complete game demo was constructed including graphics on par with the expected standard for the target platform, audio, physics, artificial intelligence and basic gameplay.

6.2 - Problems

The present implementation of many features, through both an at-the-time limited understanding of the programming language and potential design limitations, negatively impacts both performance and flexibility. Some systems run slowly unnecessarily creating objects or incurring boxing, while as a reusable objects library the code falls short heavily in reliability and clarity. Many objects expose state data they should not, or are overly reliant on the integrity of data passed to them, such as objects that store mutable reference types passed to them. This is largely down to inexperience with C# and .Net, however. Approaching the ideas with more rounded experience now would likely result in an overall better codebase.

Many features of the codebase assume a CPU-centric approach. Light, for instance, is sampled from a single point at a time, CPU side. For a low-fidelity game such as a current-generation mobile game, this is perfectly acceptable, as light is calculated for each dynamic object from a single point. On more advanced platforms such as home consoles or next-generation mobile games, lighting is expected to be sampled per-pixel for higher quality lighting. The cost overhead of doing this CPU side is that the GPU transforms the vertices, so the transformed polygonal data either needs to be transferred back to the CPU in some manner (such as copying the depth buffer back) or the same transformations also need to be done CPU side to avoid the

transfer. Either is highly inefficient. Then the computed lighting would need to be transferred back to the GPU to display. This would allow the C# lighting implementation to continue to be used, but would make very poor usage of system resources. The alternative is that such higher end platforms would have GPU implementations of the light types in their rendering pipeline, and a full deferred renderer, but this would dispense of the advantages of code reuse in the falloff system.

Implementation of the navmesh currently allows no dynamic geometry. As such, it is impossible to implement moving world parts such as lifts, doors or moving platforms. Some of the functional aspects could be added by including an accessible flag to the navmesh triangle class, replacing the neighbour references with accessors that return null if the flag is false. This would add the capability to non-destructively add and remove navmesh triangles, so paths can be changed.

Another issue with the navmesh caching system is that it cannot be used to accelerate operations that do not occur on the navmesh. Some entities, such as the third person camera, do not track with the navmesh. This means audio processing in particular cannot be done using the navmesh as an acceleration structure. For a small scale game such as this, there are not many audio sources, but the overhead could become more severe in a larger, more complex environment.

Likewise, the existing navmesh implementation cannot store dynamic falloffs. While there would not be much use for these in the context of Combat Pool, a game using a large number of dynamic lights or physics attractors would perform poorly compared to static objects. With a larger number of players, collision performance may become prohibitively slow too, so caching dynamic actors in the navmesh may also become necessary for large numbers of actors such as crowds. Subdivision of the navmesh may improve performance in these situations by reducing the number of entities per triangle.

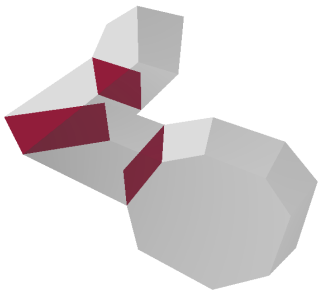
Another matter to address with regards to navmesh caching is the current approach of putting all falloffs into a single list and using type checking to determine which to use when sampling for say light. This step could be skipped by keeping a separate list for each kind of sample, perhaps using a generic interface for sampling the data and a separate type for each kind of sampleable data. [46p.129]

6.3 - Recommendations

Though the purpose of a navmesh is to aid in pathfinding through abstraction of a complex space, the example game does not perform any kind of pathfinding. Artificial intelligence is restricted to exploring the navmesh using a series of heuristics and chasing any spotted targets using a line of sight algorithm. As the navmesh is allowed to cache entities inside its triangles, this could be useful in a real time strategy game for instance to find nearby resources quickly and efficiently via a floodfill approach.

Similarly, though the acceleration structure was designed to increase the performance of few objects sampling data on a low-power platform, the same design principles could be applied to simulate massive crowds of actors on a higher end platform, minimizing the cost of collision, lighting, artificial intelligence and object avoidance by using the navmesh triangle entity lookup.

A limitation of creating the navmesh from flat surfaces is that it essentially makes the game two dimensional, with those two dimensions mapped into a 3D space. Building a navmesh from convex polyhedra (either tetrahedra or more complex geometry) would allow motion within the space, but would complicate level design and importing, as building convexes from a triangle soup and identifying how they form a representation of 3D space is a non-trivial task.



An example of how a spatial navmesh could be produced, by defining the bounds of space and dividing it into convex polyhedra using marked polygons. This is not unlike BSP in some ways, but the content is produced in a very different fashion.

7 - Bibliography

[1] – Epic Games, Inc. *Unreal Engine: Downloads* [Online]. 2008-2013. Available From: http://www.unrealengine.com/en/udk/downloads/ [Accessed: 2nd May 2013]

[2] - Epic Games, Inc. *Unreal Developer Network UnrealScript Source Files* [Online]. 2013. Available from: http://udn.epicgames.com/Three/UT3Mods.html#UnrealScript%20Source%20Files [Accessed: 27th April 2013]

[3] - Bethesda Game Studios *Elder Scrolls V: Skyrim*. Bethesda Softworks. 2011.

[4] - Bethesda Softworks *Elder Scrolls Arena* [Online]. 2013. Available from: http://www.elderscrolls.com/arena/ [Accessed: 27th April 2013]

[5] - Devrs *GameBoy FAQs* [Online]. Unknown. Available From: http://www.devrs.com/gb/files/faq.html [Accessed: 2nd May 2013]

[6] - Decker, L. DVD Writer Dustup - *Maximum PC*. November 2004. p. 78.

[7] - Epic Games, Inc. *Unreal Engine: Rendering* [Online]. 2008-2013. Available From: http://www.unrealengine.com/features/rendering/ [Accessed: 2nd May 2013]

[8] - Penny Arcade *Extra Credits: Metrics* [Online] 2011. Available From: http://www.penny-arcade.com/patv/episode/metrics [Accessed: 27th April 2013]

[9] - Carmack, J. Static Code Analysis. [Online]. 24th December 2011. Available From: http://www.altdevblogaday.com/2011/12/24/static-code-analysis/ [Accessed: 28th April 2013]

[10] - YouTube *QuakeCon 2011 Keynote* [Online] 2012. Available From: http://youtu.be/wt-iVFxgFWk?t=2m34s [Accessed: 27th February 2013]

[11] - Epic Games, Inc. *UDK Licencing FAQ* [Online] 2009-2011. Available From: http://www.unrealengine.com/udk/licensing/licensing-faqs/

[12] - Hargreaves, S. Reach Vs. HiDef [Online]. 12th March 2010. Available From: http://blogs.msdn.com/b/shawnhar/archive/2010/03/12/reach-vs-hidef.aspx [Accessed: 28th April 2013]

[13] - Epic Games, Inc. *Unreal Developer Network Designing Content For Mobile* [Online]. 2001-2013. Available From: http://udn.epicgames.com/Three/DesigningForMobile.html [Accessed: 28th April 2013]

[14] - Sillicon Graphics, Inc. *OpenGL SDK Documentation: glFlush* [Online]. 1991-2006. Available From: http://www.opengl.org/sdk/docs/man/xhtml/glFlush.xml [Accessed: 28th April 2013]

[15] - Microsoft *Microsoft Developer Network: Accurately Profiling Direct3D API Calls (Direct3D 9): Controlling the Command Buffer* [Online]. 2013. Available From: http://msdn.microsoft.com/en-gb/library/windows/desktop/bb172234%28v=vs.85%29.aspx#Controlling_the_Command_Buffer [Accessed: 28th April 2013]

[16] - Lord, K. *Optimizing with Unity iPhone, the first three things I'll do...* [Online]. April 2010. Available From: http://www.cratesmith.com/archives/183 [Accessed: 28th April 2013]

[17] - Bukket Games Blog *Managing Drawcalls on Mobile Devices.* [Online]. July 10, 2012. Available From: http://playbukketgames.com/?p=1420 [Accessed: 28th April 2013]

[18] - Hårsman, J. *why are draw calls expensive?*. [Online] 1st February 2011. Stack Overflow. Available From: http://stackoverflow.com/questions/4853856/why-are-draw-calls-expensive [Accessed: 28th April 2013]

[19] - Shimpi, A. L. *The iPhone 3GS Hardware Exposed & Analyzed* [Online]. 10th June 2009. Available From: http://www.anandtech.com/show/2782/3 [Accessed: 27th April 2013]

[20] - Nokia *Detailed specifications for the Nokia Lumia 710* [Online]. 2013. Available From: http://www.nokia.com/gb-en/phones/phone/lumia710/specifications/ [Accessed: 28th April 2013]

[21] - Chadwick, E *Beautiful, Yet Friendly Part 2: Maximizing Efficiency: Welcome to Splitsville* [Online]. 2003. Available From: http://www.ericchadwick.com/examples/provost/byf2.html#wts [Accessed: 28th April 2013]

[22] - Mirzaie S. M. Possible to get more than 20K+ triangles at 35fps on iPhone 3GS? [Online] 1st April 2011. Stack Overflow. Available From: http://stackoverflow.com/questions/5358789/possible-to-get-more-than-20k-triangles-at-35fps-on-iphone-3gs [Accessed: 28th April 2013]

[23] - Microsoft, Inc. *Microsoft Developer Network* [Online]. 2013. Available From: http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206936%28v=vs.105%29.aspx#BKMK_Hardwarerequirements [Accessed: 2nd May 2013]

[24] - Wild, J. D. *Screenshot of various kinds of falloff in the Unreal engine (V10246)*. Staffordshire University. 2012.

[25] - Unity. *Lightmapping Quickstart* [Online]. 2013. Available From: http://docs.unity3d.com/Documentation/Manual/Lightmapping.html [Accessed: 2nd May 2013]

[26] - Wild, J. D. *Standard light shapes in the Unreal Engine (V10246)*. Staffordshire University. 2012.

[27] - Wild, J. D. *The effect of using exponentiation to delinearise light falloff in the Unreal engine (V10246)*. Staffordshire University. 2012.

[28] - Sanglard, F. *Quake 3 Source Code Review: Renderer* [Online]. 30th June 2012. Fabien Sanglard's Website. Available From: http://fabiensanglard.net/quake3/renderer.php [Accessed: 2nd May 2013]

[29] - Epic Games, Inc. *Unreal Developer Network Global Light properties, Lighting Reference* [Online]. 2001-2013. Available From: http://udn.epicgames.com/Three/LightingReference.html#Global%20Light%20Properties [Accessed: 28th April 2013]

[30] - Strothotte T. & Schlechtweg S. *Non-Photorealistic Computer Graphics: Modelling, Rendering and Animation.* San Francisco: Morgan Kaufmann Publishers Elsevier Science. 2002.

[31] - The Textures Resource *Super Mario 64: Bowser In The Fire Sea* [Online]. 2001-2010. Available From: http://www.textures-resource.com/other_systems/Mario64/sheet/1210 [Accessed: 2nd May 2013]

[32] - Epic Games, Inc. *Unreal Developer Network: Lighting Reference: Lighting Subdivisions* [Online]. 2001-2012. Available From: http://udn.epicgames.com/Three/LightingReference.html#Lighting%20Subdivisions [Accessed: 2nd May 2013]

[33] - Mod DB *Engines: Quake Engine: Watchers* [Online]. 2002-2013. Available From: http://www.moddb.com/engines/quake-engine/watchers [Accessed: 2nd May 2013]

[34] - Péchard, S. *OpenGL ES: What version of GLSL is used in the iPhone(s)?* [Online]. 11th August 2010. Stack Overflow. Available From: http://stackoverflow.com/questions/3456669/what-version-of-glsl-is-used-in-the-iphones [Accessed: 2nd May 2013]

[35] - Simon, S. A realtime game audio primer. *Develop Online*. [Online]. 20th July 2012. Available From: http://www.develop-online.net/features/1676/A-realtime-game-audio-primer [Accessed: 2nd May 2013]

[36] - Ranta-Eskola S. *Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering*. [Online]. 2001. Available From: http://archive.gamedev.net/archive/reference/programming/features/bsptree/index.html

[Accessed: 2nd May 2013]

[37] - Young, S. *Twenty Sided Tale: The Truth About The BSP* [Online]. 31st August 2009. Available From: http://www.shamusyoung.com/twentysidedtale/?p=4759 [Accessed: 28th April 2013]

[38] - Baker *how do i convert .map to .bsp* [Online]. 21st April 2008. Quake One. Available From: http://quakeone.com/forums/quake-help/servers-coding/3495-how-do-i-convert-map-bsp.html#post43825 [Accessed: 28th April 2013]

[39] - Chandran S., Fang H. *Introduction to kd-trees*. [Online]. 2002. Available From: http://www.cs.umd.edu/class/spring2002/cmsc420-0401/pbasic.pdf [Accessed: 2nd May 2013]

[40] - Goodwin S. D., Menon S., Price R. G. *Pathfinding in Open Terrain*. [Online]. 2009. Available From: http://www.cgs.com.vn/upload/document/cgs_document_20100808112820.pdf [Accessed: 2nd May 2013]

[41] - Toll W. V., Cook A. F., Geraerts R. *Navigation Meshes for Realistic Multi-Layered Environments. [Online]. 2011.* Available From: http://www.staff.science.uu.nl/~gerae101/motion_planning/navmesh.html [Accessed: 2nd May 2013]

[42] - Epic Games, Inc. *Unreal Developer Network: Using Navigation Meshes: The Pylon* [Online]. 2001-2012. Available From: http://udn.epicgames.com/Three/UsingNavigationMeshes.html#The%20Pylon [Accessed: 2nd May 2013]

[43] - Bethesda Softworks *GECK: Bethsoft Tutorial Navmesh: Creating a Navmesh Triangle* [Online]. 2011. Available From: http://geck.bethsoft.com/index.php/Bethsoft_Tutorial_Navmesh#Step_1:_Creating_a_Navmesh_Triangle [Accessed: 2nd May 2013]

[44] - Perry J. & Sherrod A. & Morton M. *Essential XNA Game Studio 2.0 Programming*. Texas: Wordware Publishing, Inc. 2002.

[45] - Microsoft, Inc. *Microsoft Developer Network: SoundEffect Members* [Online]. 2013. Available From: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.audio.soundeffect_members.aspx [Accessed: 2nd May 2013]

[46] - Stahler W. & Clingman D. & Kahrizi K. *Beginning math and Physics for Game Programmers.* Indiana: New Risers Pearson Education. 2004.

[47] - Wagner B. & Meyers S. *Effective C# 50 Specific Ways to Improve Your C# Second Edition.* USA: Addison-Wesley Pearson Education. 2010.

[48] - Larry Lieberman *Microsoft Developer Network: Samples: Get To WP Mango #1 - From XNA*

Game to SL/XNA Game [Online]. 2011. Available From: http://code.msdn.microsoft.com/Get-To-WP-Mango-1-From-XNA-05fdefda [Accessed: 27th April 2013]

[49] - The Gnome Project *Gnome Dev Center: Custom Containers* [Online]. 2005-2011. Available From: https://developer.gnome.org/gtkmm-tutorial/2.24/sec-custom-containers.html.en [Accessed: 2nd May 2013]

[50] - Hargreaves, S. *Transitions part four: rendertargets* [Online]. 23rd May 2007. Available from: http://blogs.msdn.com/b/shawnhar/archive/2007/05/23/transitions-part-four-rendertargets.aspx [Accessed: 28th April 2013]