LUMENS

A STRATEGIC ACTION GAME WITH LIGHT AND PHYSICS SIMULATION, AND EMERGENT ARTIFICIAL INTELLIGENCE

FINAL REPORT

Eoghan O'Keeffe – BSc (Hon) Entertainment Systems – 08543453
Project 30 – Supervised by Dr. Kieran Murphy

May 2012

ACKNOWLEDGEMENTS

I want to thank my project supervisor Dr. Kieran Murphy for his tireless help; course leaders Karl Sandison and Rob O'Connor for their years of work for this course; the many developers, academics and enthusiasts whose work inspired and aided this project; and my friends, family, classmates, and lecturers who summoned the patience to listen to and advise me on my ideas every step of the way.

DECLARATION OF AUTHENTICITY

Except where explicitly stated, this report represents work that I have done myself. I have not submitted the work represented in this report in any other course of study leading to an academic award.

TABLE OF CONTENTS

INTRODUCTION

Lumens has been on my mind for quite some time.
I arrived at the core idea long before I had a hope of actually achieving any of it, though it was based around many areas and ideas which I was very interested in. So, it stuck with me, and by the time I had to write a design document for it, it was a fully fledged piece of work in my head; an ideal that – due to its size, complexity, and my lack of prior knowledge in some key areas – I knew I would never fully realise in the time available.
However, the process of chasing it and at least starting to drag it into reality is something I have found challenging, frustrating, educational, and rewarding. Over the course of this project, I have written detailed and aspirational accounts of this idea; delved into some fascinating and some forbidding papers on its many components; and found that Hofstadter's Law holds true as always: *it always takes longer than you expect, even when you take into account Hofstadter's Law*.
The most challenging aspects of this project have been those which I couldn't anticipate before actually getting engaged in trying to figure them out – working around the peculiarities, limitations, and intricacies of a language new to me; repeatedly reworking the structure of the project as it developed to make it work neatly and robustly; researching many different approaches to the central challenge of real-time ray-tracing, and trying to distil and adapt the best solution from them. For the most part, I enjoyed getting stuck into these efforts, learning and designing my way around the problems of realising the idea. But having gone through it, even my more modest estimations of how it would all turn out seem naïve, the real issues encountered never really considered back then – I suppose I could put it down to overconfidence in my future self's ability to overcome unforeseen difficulties. I have learned a lot, applied myself to the effort, and I am determined that this pipe-dream will eventually be realised in its full form.
As it now stands, I am satisfied to have addressed some interesting and large challenges and produced the first sights of this thing I want to produce.

The document that follows is divided into two main parts. The first – the design part – details the concept and design of the ideal *Lumens*; the second – the implementation part – shows the course the project took in trying to achieve it.
As mentioned before, the design part is aspirational – many parts exist there to show how the eventual game would be. The aspects of core importance are outlined in the opening concept section and reflected in the final estimations of deliverable targets. This represents the ideas as fully explained as possible.
The implementation part will recount the challenges encountered and the various methods and solutions attempted to address them (and, more generally, the things I found most interesting about this project); the process the project went through.

PART I – CONCEPT, DESIGN AND PLANNING

SECTION I – OVERVIEW

1 - Concept

A light-emitting protagonist navigates a dark environment, home to lurking swarms of phosphorescent entities, which are attracted to and feed on light. These predators flock passively in the darkness until they are lit; they then leave the swarm to pursue the source of light and consume it.

In order to survive and progress through the environment, the protagonist negotiates its predators in the dark, skirting the edges of the ever-moving and swirling swarm, drawing away manageable numbers of individuals and eliminating them subtly. It must strategically hunt, lure and pick apart the swarming enemy without themselves becoming prey to their overwhelming numbers.

The dark conceals the details of the environment and its other occupants, and the player is never sure of exactly what lies outside their protagonist's small pool of light – the ever-moving swarm of glowing predators can be an unpredictable and deceptive threat.

Everything in Lumens is emergent – the environments the player navigates, the behaviour of the enemies they face, and even what they see and hear – and nothing is predictable.

2 - Game Flow Summary

The player negotiates the game's core mechanism – a balanced negative feedback loop, where the more aggressive a player is, the more difficult the game becomes – in pursuit of their objectives – speedy and skilful progression.

There are two main components of this mechanism; the first is how light affects the game and its dark environment:

- Enemies hunt out the source of light when they are lit by it
- The player can only attack those within range of their emitted light

The second is the effect of the player's activity in the game:

- The range of the player's emitted light grows with each attack
- This range is only reduced back to its minimum again slowly over time
- The minimum range is increased for each enemy eliminated

The player balances their objectives with survival and risk limitation. They must get in close to the swarm in order to mount an attack; they risk attracting overwhelming numbers of predators at once; each attack increases this risk as the pool of light grows; the risk abates over time, but the goal of quickly eliminating the swarm encourages the player to continue attacking; finally, the minimum radius of light grows inversely to the number of predators remaining, keeping the overall level of threat stable as the player progresses.

This mechanic promotes strategy and controlled risk, as well as speed and skill, and makes the game more interesting, challenging, and resistive the harder the player tries. The swarm is an organic, unpredictable, and mobile enemy – the player must advance, but not recklessly. Frantic

and aggressive play is likely to result in the player facing impossible odds, whereas overly cautious and reserved play will not result in a fast completion.

3 - Feature Set

The game features:

- Artificial intelligence modelling collective, emergent flocking behaviour (Reynolds, 2001; Buckland, 2005)
- Convincing real-time light simulation, with rays, shadow-casting, reflection, refraction, caustics, and after-image, which will form the core of the visual effects
- Soft body physics, with collision-detection and influencing light rays
- Procedurally-generated levels
- Interactive, emergent audio, generated by and enhancing the game-play
- Variety of antagonists, abilities, and settings
- Support for multiple devices, with elegant solutions for smaller, touch-screen and re-orientable interfaces
- Persistence – saving and loading
- Web-accessible high scores and leader-boards

It aims to emphasise procedurally-generated features – effects, animations, physical simulations, behaviours, audio, and environments should all be largely algorithmically generated at run-time, and react to the user and the game. Pre-defined resources are kept to a minimum, and the experience instead emerges from activity within the bounds of the game environment.

4 - Genre

This is an action game with shooting and strategy elements – 2-dimensional, with a top-down or side-on view.

5 - Look and Feel

The aesthetic experience of the game is dynamic and emergent – generated according to some initial state or input reacting to the physical rules of the game environment – and there are few if any pre-created resources which are used as-is to create the experience.

Light simulation plays the central visual role in the environment – everything interacts with the rays emitted by the protagonist, and these effects combine to produce the main visual element of the game. It gives the player a strong, literal indication of their impact on the world – their visibility and vulnerability as they strive to remain hidden.
The protagonist casts an even, circular pool of light all around it from a bright, glowing coil at the center of its body. It also casts a stronger, narrow beam in the direction it is facing, which shows what it is aiming at.
Objects within the pool of light block, reflect and refract rays, casting shadows and caustics in the environment.

Significant objects (obstacles in the level and enemies) beyond that pool are phosphorescent, having some part of them that glows in the dark; they also react to the distant light, glowing brighter when facing the light.
In the absence of these lights, everything is dark and concealed.
The “living” entities are soft bodied (jelly- or raindrop-like), changing shape as they move or collide with other objects. This effect changes the way light travels around and through their bodies, and the shadows and caustics they cast as a result.
When the protagonist attacks, it fires a ropey tendril in the direction it is aiming. This tendril as long as the radius of light, and unfurls from the protagonist's body until it either hits an object or reaches its limit; it is then retracted, pulled tight and back into the protagonist's body, like a rope jerked back from its end.
Key effects accumulated from all this include the glittering, swarming predators shining and flashing as they collectively move, like bright fish in a “bait-ball”; the dramatic and interesting shadows and caustics constantly changing in the pool of light; the mysterious semi-concealed environment and its occupiers in the dark; and the elongated trails and morphing bodies of the soft bodied entities.
Colours will be kept subtle and appropriate, with only key points of interest being given a highly-saturated primary colour – such as the protagonist's light-emitting core, and the predators' glow.

The auditory aspect of the game is similarly generated in-game, and forms an integral part of the experience – it is often a sidelined aspect of games.
A simple, subtle, metronomic beat plays as atmospheric background music and a basis for the generated sounds to be layered over.
Each significant action in the game has its own simple sound, which grow louder the closer they occur to the protagonist – for example, the predators each play a sound constantly, which grows louder with their speed and angular velocity, and so changes in time with and describes their movement. Various sounds exist for entity motion, state changes, damage, attacks, and deaths; menu and HUD (Heads-Up-Display) actions; and more.
These sounds are largely simple, repeatable notes – samples from real-world instruments, or digitally synthesised notes – which can combine to produce a fusion of sound over and in step with the metronomic background beat. They react to the action in the game, and so the music describes it – they take the place of “normal” sound effects.

Finally, other considerations for the “look and feel” of *Lumens* include simple and absolutely minimal menus and HUD – everything will preferably appear on the same screen, but slide in and out of view as needed – with simple and appropriate typeface, reacting to the light of the game screen beneath; haptic feedback (used sparingly) during significant events, for game controllers and other supporting devices; and indicative guiding visual cues to aid user interaction on touch devices – consisting of a subtle, glowing ring appearing around the positioning finger, and a glowing anchor point on that ring for the directional finger, which follows the angular motion it traces (see *Section IV: 1.2* for a detailed explanation of the touch-screen input system).

6 - Target Audience

Lumens is a game particularly suitable for short bursts of play, with strategic, non-repeating scenarios and a strong, recognisable product identity. It is aimed at players looking for either quick diversions or longer sessions of use; who enjoy action and strategy games, unusual ideas, and memorable "interactive experiences".

7 - Project Scope

This is an individual project, with an agile software development approach. Each version produces a new component of the project's modular architecture, as described in *Section VII*; it was expected, however, that these goals would adapt and change over the course of development, so they represent initial, speculative goals for planning purposes only. The expected minimum goals to be reached by the end of development are outlined as *Iteration 0.1* in that section, with further goals outlined in the subsequent iterations.

By the end of the first phase of this project, I aimed to have reached *Iteration 0.1* – that is, achieving a demonstrable working prototype of the core game-play functionality, and development of the system components needed to achieve it. This involves the design of a suitable architecture for the system – which will be explored in the next document – and the development of the basic components of that system – including AI, input, and rendering with simple shapes, but not light simulation, level generation, etc. – as well as the investigation of and experimentation with each of the used software components and the first target platform.

In addition to the work involved in designing, programming and developing the components of this project, there is work which must be carried out in research and investigation of those unknown components and areas which I was encountering for the first time here, or had encountered little before. Numbering among these are real-time light simulation and soft body physics theory, approach, and implementations; the production of full engines to handle game logic, physics, rendering, and artificial intelligence; platform-specific concerns such as capabilities, user-interface, programming interfaces, development and production tools, and enhancements; and development of efficient systems for fast tweaking and balancing of the system, and platform abstractions, where possible. This investigative work added substantially to the process, and I tackled many new areas through it.

SECTION II – GAMEPLAY AND MECHANICS

1 - Game-play

1.1 - Game Progression

- The player progresses through stages
- Each stage is generated algorithmically, and becomes progressively more difficult, according to the level generator
- When a player completes the objectives at a stage, they transition to the next generated stage
- This process may carry on indefinitely until the protagonist is eliminated
- The player then sees their score in a scoreboard, and may begin again or load a previously saved game

1.2 - Objectives

- The player must eliminate the predators occupying the environment
- The faster they achieve this, the better their score
- Each enemy they eliminate gives the player a set amount of points, which is increased by a factor according to the score multiplier system
- A successful attack starts the multiplier, which immediately begins reducing with time
- Each attack following it has its reward points multiplied, and adds a little to the multiplier
- Multiple attacks and attacks on more challenging predators are worth more points and add more to the multiplier
- The multiplier may increase to quite a high level, but may only reduce by a much smaller amount over time before it vanishes – this smaller amount is increased at a fraction of the speed of the multiplier increase
- The multiplier system promotes speed and constant activity to keep it going
- Counterbalancing these goals is the goal of survival – although speedy completion and maintaining the multiplier promote speed and activity, the more active the protagonist is, the bigger a target they are to their predators
- The game may also be enjoyed as a goal-less experience
- Further objectives will be explored in any additional modes that may expand the game in later iterations

1.3 - Challenges

- The swarm of predators and the environment they are in present challenges to the player
- The predators hunt the player just as the player hunts them
- Depending on the type and number of individual predators in the swarm, it behaves in different ways as a whole
- The swarm as a whole moves in dynamic and difficult to predict ways, swirling around the environment, presenting a moving, shape-shifting target and threat to the player

- As the player's activity increases, they present a larger target to the swarm
- The dark environment partially conceals and distorts information to the player
  - Predators have different visibilities in the dark, which can misinform the player (see *Section III: 3.2* for further details on each type's behaviour)
  - The obstacles in the environment also have phosphorescent surfaces, but only those facing the player glow in the dark, so the full environment isn't always visible
- Procedurally generated stages present additional challenges
  - Players may never encounter the same one twice, meaning they cannot "learn" a particular stage, and are always "blind" to the layout of the environment and what it contains
  - The level generator produces progressively more challenging stages, varying environmental complexity, size, the number and types of predators, and other factors
- In all, the dark conceals threats; the predators' various natures mislead and surprise; the ever-changing swarm outnumbers, entraps and surrounds; the player's own activity makes them more vulnerable; and the stages are unpredictable and progressively more challenging

1.4 - Game-play Flow

- The player enters a new environment, casting a small radius of light and appearing at a safe distance from the predators dwelling there
- They immediately begin navigating the environment and seeking out the swarm, exploring the darkness
- The swarm moves obliviously in the dark, its members unaware of the player until they are lit
- The player moves towards the swarm to begin eliminating predators, trying to lure away and attack small numbers of them without becoming overwhelmed
  - The number of the predators' glowing cores and their movement offer a clue as to what type they are, how difficult they will be to defeat, and how large a reward they will leave if beaten – in addition, the number and brightness of the cores they have indicates how much health they have remaining
  - Although only the predators lit will attack the protagonist, others may follow them, also become lit, and join the attack – the more attracted at once, the greater their pull on the rest of the swarm
  - The player will usually try to attract as few predators as possible at one time
  - As the player's radius of emitted light grows with their activity, it becomes more difficult to avoid accidentally attracting unwanted extra attention – the light cast is a growing, uniform shape negotiating an irregular, changing, difficult to predict shape at close range, which may surround or cross it at any time

        - The player may also succumb to the tricks of the environment and predators, and become trapped in an unexpected series of obstacles, or be caught off guard by deceptive types of predators (see *Section III: 3.2*)
        - The player may also target special members of the swarm, which carry collectible items that aid the player in certain ways
            - These predators have an additional glow, colour-coded according to the collectible item they are carrying, and differentiating them from their flock-mates
            - The types and effects of these collectibles are detailed in *Section III: 2.4: (a)*
            - This introduces a more targeted and selective hunting behaviour – if a player has taken a lot of damage, for example, they may stalk a predator carrying a healing collectible (identifiable by their secondary green glow) and wait until they are exposed at the edge of the swarm to strike, to reduce the risk of being attacked by others while injured
- If the player lures away an enemy, they attack one another
    - The enemy chases the protagonist down, trying to reach it and attack it at close range
    - The player fires at the attacking predator, and moves to prevent it getting close enough to harm them – they can outmanoeuvre most enemies, so attacking while backpedalling, and hiding behind obstacles may be useful tactics
    - This is often not a one-on-one confrontation, as many predators may attack at once
- The protagonist also has some special moves at the player's disposal, which all give temporary added power and advantage, but come at the cost of greatly increased profile after use (see *2.3: (a): Enhanced* later in this section for details)
- If the player is finding it too difficult to negotiate the swarm with a profile that's too high, they may try and retreat to allow it to reduce over time
    - The profile reduces faster in inverse proportion to the current score multiplier
    - If they choose to retreat, the player sacrifices their score in both time and their multiplier bonus
- When the player has eliminated the swarm, or completed the stage objective (eliminating every member may prove slow and tiresome, so the objectives or mechanisms may be altered to prevent this after play-testing), they transition to the next stage that the level generator produces
- This process may repeat indefinitely until the player is defeated (but, again, this may be altered at a later stage, according to play-testing results)
- The player then sees their final score, and may proceed from the beginning or a save point
- Currently, the player gets one "life", and when they are defeated, the game ends – but this may be revised to the following multiple life system later if necessary

- When defeated, the protagonist's light is extinguished, and the player moves their glowing ember to a safe starting position
- After a short time, the protagonist's light reappears at its starting size at the new position, and play continues
- This behaviour is similar to the "beam" special move described in Mechanics: Combat: Enhanced later in this section
- This is allowed to happen only a few times, before the protagonist is permanently defeated

2 - Mechanics

2.1 - Physics

(a) Light

- Real-time light simulation forms the visual and thematic core of Lumens
  - Backwards Ray-Tracing is the desired target for this simulation (Owen, 1999; Wikipedia, 2011), but other approaches – such as Photon Mapping, Shadow Mapping, or Metropolis Light Transport, and others – will be considered to achieve suitable performance and the desired effects
  - The desired effects of this simulation are rays and shadows, with good accuracy; reflection and refraction, to a low depth (number of times a ray can bounce around a scene); and caustics, to a low degree of accuracy (this will require some additional methods – such as bi-directional ray-tracing or mapping – and is a purely aesthetic part of the game)
  - After-image will be achieved in a final additional pass over the fully rendered scene, where the results will be processed to find the parts of the image brighter than a minimum threshold, and these will be persisted and redrawn for some time in subsequent frames
- Only particularly bright effects will cause after-image, to avoid cluttering the screen

(b) Particles

- Simple force-based particle physics, primarily for use in visual effects

(c) Rigid Bodies

- Force-based simulation of non-deformable objects – such as the environmental obstacles – and their motion

(d) Soft Bodies

- Force-based simulation of deformable bodies – objects whose shape varies with external forces and tends towards a stable shape

- The body's particles exert forces upon their neighbours until a stable state is reached, but may be affected by external forces, making the system unstable again
- Need to support at least two applications – "blobby" closed shapes and "ropey" open ones – to a visually satisfactory level

(e) Collisions

- Full kinematic collision engine supporting soft bodies, rigid bodies, particles, geometry, and light rays

2.2 - Movement

- 2-dimensional, within the constraints of the physical laws of the environment and the entity's own propellant and turning forces
- The protagonist is directed by input from the player (see *Section IV: 1* for details specific to each platform)
- The protagonist may move and aim independently and in any direction
- Other entities are moved according to their AI
- Primary movement is done on the center of an entity, and the body of the entity then moves in relation to this
  - For a soft-bodied "blobby" entity, the particles defining the body's edges are pulled by a force acting towards the moving center, but do not instantaneously move with it
  - For a rigid-bodied entity, the body moves along with its center
- The particulars of movement vary for different entities and an entity's different states (see the *Behaviour* parts of *Section III: 3* and *Section V* for details)
- When the viewport (the area of the screen in which the player can see the game) is smaller than the environment, the viewport follows the protagonist's movements until reaching the limits of the environment – it is bound by a "spring" force with a maximum threshold to the player's position, and collides with the edges of the environment

2.3 - Combat

(a) Protagonist

– Normal

- The player adjusts their aim – indicated by a strong beam of light emanating from the protagonist's center along its heading – towards enemies in range
- When they attack, the protagonist fires an unfurling soft bodied tendril in the direction they are aiming
  - The leading particle of this tendril is directed, and the trailing ones are pulled along, and subject to the soft body physical rules governing the tendril

        - When the leading particle either strikes an entity or obstacle, or the overall length of the tendril reaches its maximum (the radius of light cast by the protagonist), the tendril is retracted – the last particle is pulled back, pulling the next ones, until it reaches the protagonist's center, then the next one is pulled, and so on
        - In order to ensure quick game-play, this all happens very quickly, and the effect of the attack is applied as soon as the leading particle strikes something – the unfurling and retraction of the tendril trailing behind it is a visual effect, and does not take precedence over usability
    - The player may execute multiple attacks in quick succession – there may be multiple tendrils extended at a time
    - If an attack strikes an entity, a collision and physical reaction are applied to both, and if it strikes an antagonist, damage is dealt to it
    - Every attack increases the radius of light cast by the protagonist
        - If the attack struck an antagonist, the increase is smaller, rewarding players for accuracy
        - If an antagonist is eliminated, the minimum radius is increased in proportion to those remaining

- Enhanced
    - There are certain enhanced modes of combat the player can activate temporarily
    - All can only be active for a short time, take a long time to become available again, and cause a large increase in the radius of cast light after use
    - They are all activated by the player and held in this active state until the player releases them again, or until the maximum allowed time is exceeded
        - In order to correctly interpret the player's input, there may be a very slight delay before the mode is activated (see *Section IV: 1.2* for details)
    - While in an enhanced state, the protagonist's element (the filament encircling its core) proceeds towards a different colour over time, a sound grows in volume and tempo, and there may be some other effects – these are the player's warnings of the limited and expiring time they have left in this mode, as well as the growing penalty to follow
    - Upon exiting these states, the penalty in increase of the radius of light and the period of recovery for that enhancement are applied in proportion to the amount of time spent in the enhanced state – the light grows slightly beyond its bounds, then retracts to the new

radius (attracting nearby predators and get the game under way again quickly)

- The protagonist contains glowing cores for each mode, which darken and brighten according to how fully charged each of the corresponding modes is, with the same colour code as that mode

- Beam
  - Can be used as a way to get out of a tight spot and evade pursuing predators, as well as an offensive weapon against them
  - The protagonist's light shrinks and disappears, leaving only the protagonist's glowing core and element visible – as a result, the player is no longer pursued by predators
  - While the protagonist is in this unlit state
    - It may be directed as usual, but receives a speed boost – the player may wish to find a safe spot in the environment to relocate to
    - When it makes contact with predators, it is unharmed and severely damages them – the player may aim for predators in order to eliminate them, and the usual rewards for attacks are applied
    - Its element colour proceeds towards red
  - Upon exiting, the radius of light suddenly and rapidly reappears and expands

- Repel
  - The protagonist emits a pulse from its core, causing a repellent force on all objects within its range – a defensive weapon
  - The protagonist's element colour proceeds towards purple
  - Upon exiting this state, the repellent pulse is released from the protagonist's core, and grows rapidly outwards, the same shade of purple as the protagonist's core was upon exit
    - The pulse's final size and force are proportional to the time spent charging it up in this state
    - The force is stronger towards the center of the pulse
    - It applies this force in the direction away from the protagonist to any objects caught in it, and any predators receive some proportional damage
    - There may be some other visual and auditory effects
    - The radius of light simultaneously grows to its new size
  - This is a pretty standard attack in this genre of game, so a more interesting version may be a “black hole” effect

- The screen becomes a negative image of itself, and a vortex grows steadily from the player
- This vortex sucks everything around it to its center and destroys it
- All predators flee from it
- Upon exiting this state, the vortex implodes, and a repellent pulse explodes at the same time, like the one described above, pushing predators away from the no longer invulnerable player

- Range
  - The protagonist's radius of light shrinks and the length of its aiming beam grows inversely by the same factor
  - Light expands around its aiming beam into a conical (or triangular, since this is viewed in 2 dimensions) spotlight
  - While the protagonist is in this state
    - The player may fire with the temporarily extended range
    - The damage done by each attack is increased, but the tendrils do not extend any faster, so take longer to reach more distant targets
    - The reduced temporarily reduced radius lowers the profile of the protagonist – though the spotlight also attracts predators, they are attracted into the line of fire
    - The protagonist's element colour proceeds towards green
  - Upon exiting this state, the usual penalties are applied, and the protagonist's light and range return to normal

(b) Antagonists

- Predators attack in a similar fashion to the protagonist
  - They aim towards their target and fire proboscis at it, then withdraw them
  - These proboscis differ visually from the protagonist's tendrils, being a pair of simple extensions coming from their body, pulling the edges of the soft body, and having sharp, pointed ends
  - Successful strikes cause damage to the protagonist
- After a successful strike, the predator pauses for some time before attacking again – as though pausing to digest their food
- Predator combat uses variations of this system, depending on their type, state, and statistics (see *Section III: 3*, and *Section V* for details)
- Aiming and movement are handled by the predator's AI, as are attacks and special behaviours

- The particulars and constraints of their attack range, speed, recovery period, damage, etc., are defined by the statistics of each predator
- If the protagonist is defeated, it emits a sustained, large burst of light
  - The predators in range cease attacking and begin a feeding frenzy
  - Each "bite" is similar to an attack, but at zero range, and reduces the size of the light and the protagonist's body

2.4 - Interactive Objects

- There are other objects in the environment with which the entities and environment react in various ways

(a) Collectibles

- These are special items which provide a recovering boost to certain statistics of the protagonist, and are identifiable by their colour-coded glow
- They appear larger depending on the size of the boost they offer
- They glow, and float around the environment, just like a particle
  - They are attracted towards entities that pass nearby
  - Predators also seek them out, and may pick up loose collectibles
- When carried by predators
  - The predator carrying them is identifiable, as the item's glow remains visible – this allows the player to target them when the item is particularly needed
  - The predator releases the item when destroyed, and it then floats around the environment
  - When a predator is created already carrying the item, the size of the boost is proportional to how difficult the predator carrying it is to defeat
- In order to collect them, the protagonist may move into contact with them or hit them with its proboscis
- When collected, a sound is played and some particles are released

– Healer

- Repairs some of the damage done to the protagonist
- Glows green

– Score Multiplier Booster

- Applies an instant, larger boost to the player's score multiplier
- Glows yellow

– Light Dimmer

- Applies an instant, larger reduction to the radius of light cast by the player
- Glows white

- Enhanced Combat Booster
  - Applies an instant, larger boost to the recovery time for all of the protagonist's enhanced attacks
  - Glows purple

(b) Atmosphere and Particles

- Particles float around the environment; observe the physical, light, and collision rules of the environment; and may have a limited lifespan
- When significant events occur – such as a successful strike or consumption of a collectible – there may be some coloured particles released, to indicate the event visually, show the point of contact, simulate debris, and so on
- There may also be some semi-transparent atmospheric dust particles, which have no limit to their lifespan, and are pushed around entities as they come close to contact, as though affected by the air around them

2.5 - Environment

(a) Interaction

- The environment contains a set of obstacles, which no other entities can penetrate – the boundaries of the game
  - All bodies in the game collide with the environmental obstacles
  - Rigid bodies and particles collide plastically with the obstacles
  - Soft bodies squash up against them until their centre collides and stops, then elastically bounce away
- Light rays intersect with them too
- They have infinite mass, and are immovable
- Aside from these obstacles, the environment also consists of a planar backdrop, which takes up the entire area of the traversable space
  - It provides a background to the game, and is particularly important for making the light rays striking it visible – without a backdrop to project upon, the shapes of shadows and caustic patterns could not be made out
- This background only interacts with light rays

(b) Progression

- The environmental obstacles can just about be made out in the dark, reflective, glowing points on their surface reacting to direct light from a distance – the side facing the protagonist can be determined, but everything else is concealed
- The player may manoeuvre around, hide behind, and shield their protagonist's light with the environmental obstacles
- Artificially-controlled entities aim to avoid these obstacles

- As discussed in part *2.2* earlier, the viewport moves freely around environments larger than it, and collides with its edges, limiting the view to the traversable area

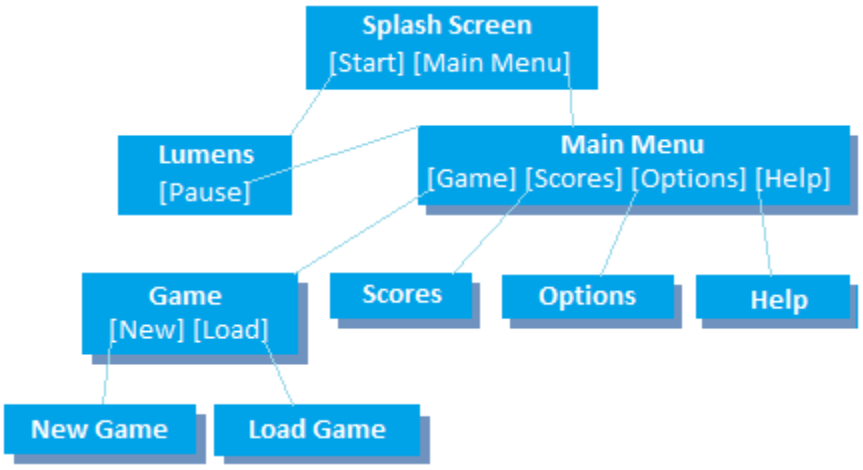
(c) Stage Generation and Transitions

- Stages in Lumens are generated procedurally, by a level generator module which produces them according to a set of criteria designed to ensure the level is playable
  - There cannot be too many obstacles crowding the space
  - Gaps between obstacles must be large enough for the player to pass through
  - To make the level visually appealing and interesting, functions which produce interesting and naturalistic graphical effects are used in the generation process – fractals may be used; or a generated field of 3D objects, which the protagonist moves along one dimension of, and the obstacles consist of the cross-section of these objects along the plane at that point (like an MRI machine); or an entity at least the size of the protagonist may carve out the level, which is initially one large obstacle (like an earthworm), and be guided by various influences (random walk, steer along the edges of obstacles, increase and decrease radius, etc.) which could be tweaked to generate different kinds of stages (maybe by a genetic algorithm)
    - This last method is desirable because it combines the generation and testing for usability of the stage into one step – (if the entity is at least as big as the protagonist, the stage will always be navigable)
- Transitions from the completed stage to the next one occur while it's being loaded
  - After a player completes a stage, the transition is ready to be started upon the player's prompt
    - The protagonist's light is more vibrant and colourful, and pulses and scatters
    - The player's score and other details are displayed over the viewport
    - The player may direct the protagonist as usual in the background, and if left alone, the protagonist wanders around alone (see *Section V: 2* for details on this wandering)
  - If the player chooses to advance to the next stage, the score information disappears, the level begins to load, and the transition begins
    - The protagonist flashes a burst of light, and suddenly moves at great speed along the axis pointing away from the camera, leaving the level's image pulled and distorted at the point of departure
    - A trail of light and motion blur effects trail behind the protagonist, giving a sense of speed and movement from one place to the next

        - The player can still guide the protagonist around the viewport in this state, to alleviate boredom from any long waits
        - If it proves suitable, the stages will be based on a large fractal, and the protagonist will move towards another region of it upon transition, which is visible as they zoom towards it – in this case, the player may direct the protagonist to some desired area of the fractal
    - When the level is loaded, the protagonist loses speed and light intensity, and arrives at the new starting position
        - The level becomes visible as they approach, giving the player a brief overview of its layout and occupants, and allowing them to pick out a suitable starting point
        - The starting point is anywhere the player chooses, within the bounds of the environment, and not inside an obstacle
        - The protagonist lands at this position, distorting the level image upon impact and severely damaging any predators in the way
    - After a brief pause, play begins again

3 - Screen Flow

- The number of screens is minimal in *Lumens*
- All information is overlaid on the viewport, with the game visible and running (paused during game-play) in the background, as described in *Section IV: 2.2*
- This section deals with the progression of screens over the course of the game



- Splash
    - The protagonist moves around an empty environment, with the title Lumens used as the background material, slightly too large to be lit all at once by the protagonist's normal light
    - At first start, its light is expanded to highlight the full title

- The player may direct the protagonist or leave them to wander around, and can expand the light cast to highlight the title at any time
- When the player chooses to move on, they may perform any short input (tap a key, button, or the touch-screen, as opposed to holding down) to move straight into the most recent saved game (Lumens screen), or alternatively press pause to access the main menu
- Menus appearing over the splash screen do not cause the background action to pause while they are open, unlike during game-play, and the protagonist wanders as usual

– Lumens
  - Any menu overlays disappear, and the splash screen briefly remains, before the protagonist transitions as usual to the starting point of the level and play commences
  - See *Section II: 2.5: (c)* for details on transitions between stages on this screen
  - At any point, the player may pause the game, and the pause screen will be presented

– Main Menu
  - From here, the player is presented the options to proceed to the game, scores, or help screens

– Game
  - The player is presented options to proceed to the new game or load game screens

– Scores
  - The player is presented with their scores and, if online, those of other players
  - These scores may be in ascending or descending order in various categories, as described in part *4.2* below

– Help
  - The help screen is a scrolling screen with a navigation sidebar, which details all of the areas of the game which the user may require information on

– New Game
  - If it is suitable to keep multiple saved games, the player may choose a name for their new game and proceed to start it
  - If the name already exists for another saved game, they are prompted to overwrite it or change the name – if they overwrite it, they then proceed to start their new game
  - Otherwise, they simply proceed to a new game, overwriting their existing single game, without this screen

– Load Game

- If it is suitable to keep multiple saved games, the player may choose a game to load, and then begin playing from the saved point
- Otherwise, they simply proceed to their existing single game, without this screen

4 - Persistence

4.1 - Saving and Loading

- Depending on the outcome of testing of suitability and requirements, Lumens may either support a single persisted instance of a game, or multiple ones which can be managed by the player
- General accumulated statistics and information relating to level generation must be stored in a persisted file
- The persistence system uses a platform-independent format to save information

4.2 - Scores

- Scores are accumulated over the course of a game, and stored when a game is ended or saved
- If an internet connection is available, the player's scores are synchronised with the global online scores – a limited number of global scores may be pulled down at this time as well
- They may be viewed at any time, and ordered according to various criteria – such as overall score, time spent playing, enemies killed, number of deaths, kill-death ratio, etc. – so the necessary information must be gathered and stored over the course of a game

SECTION III – THEMES, SETTING, AND ENTITIES

1 - Themes

Lumens focuses on a few central ideas and themes:

- Light and revelation, darkness and concealment
  - The player should be in suspense, never quite sure of what they might stumble upon as they cut through the dark – think of the feeling you have walking in the dark, before your eyes adjust, anticipating a trip or collision
  - They should be surprised by what they thought they saw in the dark, revealed as some deception in the light
  - The properties of light are the primary visual focus of the game (or those properties highlighted herein)
- Stalking, luring, and finely-balanced competition for survival
  - The player carefully hunts a prey that could easily turn on them and destroy them in kind – they are prey too
  - They must be cunning and react quickly to pick apart a threat much larger than themselves
- Negotiating the communal behaviour and overwhelming aggression of the swarm
  - A swarm behaves like an organism, the individuals' collective movement changing it in unpredictable ways
  - Managing and negotiating numbers which behave like this is a challenge requiring constant alertness and responsiveness
- The mindless attraction of simple creatures to light
  - This is the idea that began the whole project – watching a moth repeatedly throwing itself against a light bulb, thinking of huge swarms of insects, and cave-dwelling troglobites which have lost their skin pigment and sight in adapting to total darkness
- An organic and unpredictable environment
  - Ideally, everything in Lumens should be generated procedurally, emerging naturally as the result of initial conditions passed into an environment of defined rules, with unpredictable outcomes (dynamical systems and chaos)

2 - Setting

2.1 - Elements

- The obstacles contained in the environment are 2D shapes, forming the boundaries of the game
  - They are immovable, and all entities and light rays collide with them
  - The player may be surprised and trapped by an unnoticed obstacle emerging from the dark, or a misjudged shape turning out to be a dead-end
  - They may also be used strategically
  - The protagonist can be hidden from the swarm by using them to shield its light

        - The predators can be funnelled, their movements better predicted, and kept at bay by the obstacles
    - There is a 2D plane, which acts as a backdrop layer
        - Its reaction to light is the main point of interest – acting as a surface for light to be projected upon, and important for making the more interesting light effects (shadows and caustic patterns) visible
        - It does not react with any entity in the game, only light
    - Atmospheric dust particles also float around the environment, picking up rays when lit

2.2 - Look and Feel

- The setting is simple, abstract, and pitch black without the protagonist's light
    - The environmental obstacles define its shape
    - They are 2D, and made up of irregular and complex-looking surfaces – reminiscent of craggy rocks
    - These shapes can just about be made out in the dark, by glowing points on surfaces directly facing the protagonist (and not blocked by other obstacles)
    - The player should feel unsure when navigating the obstacles, whose glittering edges only give small clues the of their shape in the dark – like they are wandering around in a dark room littered with objects, and cannot quite make anything out, so explore cautiously with their hands outstretched, with the frequent feeling of being just about to walk into or stumble over something
- The backdrop plane is simple and visually unobtrusive
    - It may be an abstract bump-mapped surface, or reveal the subsequent stages (as if through a foggy window), depending on the level generation strategy used
- Atmospheric particles vary in size, are transparent, and appear bright when lit
    - They give the effect of dusty air or cloudy water

3 - Entities

- Note that the statistics offered in the Statistics subsections following are intended as rough rankings of the entities' abilities relative to one another (5 is high, 1 is low), and are not representative of any final balancing or even any applicable unit of measurement

3.1 - Protagonist

(a) Behaviour

- The protagonist is directed by the player
- It may move and aim independently and in any direction
- Attacks, both normal and enhanced, are subject to their own delays after use

- Pending the results of testing, fully automatic firing may be used (where the player can hold the fire command, and attacks occur continuously in sequence, with the delay between each one)
- During certain parts of the game (such as on the splash screen), the protagonist may wander around with no particular aim

(b) Statistics

- Speed: 4 (the maximum speed it can reach)
- Agility: 5 (linear and rotational acceleration)
- Resistance: 4 (the amount of damage it can take before dying)
- Damage: 3 (the amount of damage it deals when attacking)
- Range: 4 (the range of its attacks)
- Attack rate: 4 (the speed with which it recovers from attacking and is ready to do so again)
- Size: 3 (its overall size)
- Swarm influence: 5 (the degree to which predators are attracted towards it – for predators, this influences how closely the swarm follows them)

(c) Look and Feel

- The protagonist has a circular soft-body, which is normally a “frosted” white colour
- Its core is a glowing gold element (like a light bulb element), surrounding the source of its aiming beam which comes from its center
- This core is the source of its bright white light, its gold aiming beam, and usually the only source of (ray-traced) light in the environment
- Its other glowing cores correspond in colour to each of its enhanced combat modes (red, purple, and green), fade and grow brighter according to their level of charge, and are semi free-floating in its body, located towards the rear and either side
- During attacks, it fires tendrils from its body, which meet seamlessly with the edges of its form and are the same white in colour
- These tendrils are ropey, soft, and thin, with a small node at the end
- They are fired and withdrawn as described in *Section II: 2.3: (a): Normal*
- The protagonist moves with great agility and good speed, and should feel very responsive to control
- Upon death, the protagonist's glowing cores fade away, its element becomes a dull glowing orange, its aiming beam vanishes, its central core explodes, and a huge, sustained burst of light is released
    - While the predators are in their feeding frenzy, the protagonist's body reduces in size with each bite
    - When the light is extinguished, all that's left is the protagonist's element floating around the environment

3.2 - Antagonists

- All predators observe a variation of the same set of common behaviours
  - When unlit, all follow the swarm of their fellows according to the rules outlined in *Section V: 1*
  - They also wander, so that they are not stationary when alone, and to add further unpredictability to the swarm's overall behaviour
  - When lit, they all attack the protagonist
  - None are affected by the protagonist's cast light beyond its range – that is, they do not see it from a distance, but rather “feel” it upon contact
  - They do not have memories – when lit and then unlit, they simply return to their wandering or swarming state
  - Unlike the protagonist, they cannot move and aim independently, and must turn to face their targets
  - Just before striking, they betray a small clue as their proboscis emerge slightly, giving the player a chance to evade the attack by moving out of the way before they can turn to face them again, in which case they strike and miss – the delay is proportional to their attack rate
- Differences in this behaviour arise either from differences in their statistics, or explicit differences in their AI
  - They all have common traits while also being very distinct from each other, giving the impression of being different varieties of the same species, or the same species at different stages of development
  - Soft bodies with a slight purple-red hue
  - Containing semi-free-floating phosphorescent core(s) (the number proportional to their current health)
  - They have red spot(s) of pigment on their skin (the number proportional to their initial health), with generated shape and laid out symmetrically along its rear-front center-line

(a) Fly

– Behaviour

- The fly is the basic and most common type of predator, and all of the other predators are based upon it
- It follows the swarm when unlit, and pursues the protagonist when lit, attacking when in range and aiming correctly
- It is small and weak, but exists in very large numbers, is the fastest and hard to hit or outmanoeuvre, and because of its limited range, will get in very close to attack – many of these will smother the protagonist

– Statistics

- Speed: 5

- Agility: 3
- Resistance: 1
- Damage: 2
- Range: 1
- Attack rate: 3
- Size: 1
- Swarm influence: 2

– Look and Feel
  - The simplest and smallest of the predators, it is a soft-bodied, elliptical entity with a slight hue
  - It has a single pigment at its mouth, and contains a single point of phosphorescence
  - Its attacking proboscis are short mandibles

(b) Hunter

– Behaviour
  - The hunter is only slightly affected by the swarm; the primary influence on its movement is actively seeking out the protagonist
    - It will separate from the main body of the swarm and search the surrounding area for the protagonist
    - It moves towards likely places, making forays along the environment, following the obstacle walls, and making searching criss-cross patterns of movement in open space
    - It will not leave its swarm entirely, but rather acts as something of a scout, staying at the periphery and making sorties further away, but trying to return and catch up when it strays too far
  - It is a challenging opponent on its own and in groups, intelligently searching for prey, and attacking with speed, agility, and evasiveness when it finds its prey
  - It is more intelligent than the others, and shows this in its searching and attacking behaviours – also, it can be considered a playground for creating smart AI within the limitations of the game rules
  - It can, however, be more easily drawn away from the swarm, and eliminated on its own

– Statistics
  - Speed: 3
  - Agility: 4
  - Resistance: 2
  - Damage: 1
  - Range: 3

- Attack rate: 5
- Size: 2
- Swarm influence: 1

– Look and Feel

- The hunter is – like the others – soft-bodied, but much firmer, not deforming much, with a strong hue, and has a slightly triangular shape, with twin mandibles forming two corners at its head
- It has two pigments towards its head, and contains two glowing points
- Its proboscis come from between its mandibles, and are a long, deep-red, tongue-like thread, similar to the protagonist's, but thinner, moving more rapidly and directly, and with a more evident seam where they join at its mouth

(c) Strobe

– Behaviour

- The strobe follows the swarm as usual when in the dark, but its movements are difficult to track
  - Its glow periodically disappears and reappears, fading in and out of view
  - It changes direction more than the other predators, and is more likely to wander off and appear somewhere unexpected when not glowing
- When lit, it presents an indirect but significant threat if members of its swarm are nearby
  - It is slow and cautious, skirting the edges of the light and only reluctantly attacking the protagonist itself – when it does attack, it makes a deliberate move towards its target to get in range, then tries to retreat back to the edge afterwards
  - Rather, it retransmits the protagonist's own light from its own body, drawing other predators towards the protagonist
    - Its recast light has its own range and uniformity, and upon being lit, it gradually begins increasing these
    - It is most dangerous when at the edge of the light, attracting its flock-mates and protected by their bodies as they rush to attack the protagonist, continually increasing the protagonist's profile, bringing large numbers in to attack, and out of reach – attacking by proxy
    - If the swarm contains many of them, a chain of light could also be formed
    - Its maximum and minimum ranges are decide by the protagonist's own ranges
- It is a self-preserving and strong opponent, and a leader sending others to do its work

- Statistics
  - Speed: 2
  - Agility: 1
  - Resistance: 3
  - Damage: 4
  - Range: 2
  - Attack rate: 2
  - Size: 4
  - Swarm influence: 3

- Look and Feel
  - Its body is fat, loose and irregular, a kind of deformed version of the protagonist's circular light-emitting body, with a medium-weak hue
  - It has 3 pigments on its skin – one at its mouth and two peripheral ones on fins to either side of its body – and three glowing cores
  - The light it casts is a duller, more orange colour than the protagonist's, and its body has a very slight red hue
  - Its proboscis are a pair of thick, long limbs emerging from either side of its mouth, and not the stretchy threads of some others

(d) Behemoth

- Behaviour
  - This is the hidden boss, an unwelcome surprise in the dark
    - It moves slowly, surrounded by the swarm due to its strong influence on them
    - In the dark, it's difficult to tell if a cluster of glowing points is a group of individuals, or one large behemoth surrounded by flies – like seeing two headlights in the dark, thinking they are two motorcycles side-by-side, and discovering they in fact belong to a truck
    - As a result, an unwitting player may stumble across a big challenge unexpectedly
  - When lit, it is a big, slow, powerful fortress in attack
    - It cannot turn or move very fast, and its emerging proboscis just before an attack are a quite visible warning
    - However, it causes a great deal of damage when it strikes successfully with its proboscis
    - It also has some other strategies, subject to their own delay and range constraints
      - It is cannibalistic, and will occasionally consume other predators when damaged

- This absorbs their health, glowing cores, and any collectibles they were carrying
- The Behemoth turns towards an unsuspecting ally and pulls them into its mouth
- During this time, it leaves the protagonist alone
- If the player manages to destroy the other predator before the Behemoth slowly finishes swallowing it, the effect is prevented, and the player gains an additional boost to their score and multiplier, and the event is recorded in the score system

- It has a secondary venom-spitting attack
  - This is more rapid and has greater range, but is much less damaging than its primary attack
  - Again, its proboscis betray the onset of this attack, moving back away from its mouth to allow the venom to be fired

- It relentlessly advances as long as it is lit, and brushes off most attacks easily

- It is the apex predator of the swarm, and is best faced evasively – as with any predator, but doubly important here, since it takes far more time to kill it
  - Anticipating its attack cues is hugely important in dealing with the behemoth, as successful strikes are so damaging that dodging them is key to survival
  - Because of how large and cumbersome it is, use of the environment to evade it is an effective and important strategy – narrow gaps can be very helpful obstacles
  - Outmanoeuvring it and trying to stay out of its attacking line are hugely important
  - Leaving and returning to the fight may also be a necessary tactic
  - Enhanced attacks and collectibles may be life-saving aids
  - Finally, the Behemoth will rarely be alone, so even if the player can easily outmanoeuvre it, whether they can do so when faced with other incoming predators is another question

– Statistics

- Speed: 1
- Agility: 2
- Resistance: 5
- Damage: 5
- Range: 5
- Attack rate: 1
- Size: 5

- Swarm influence: 4

– Look and Feel

- The Behemoth's body is soft towards the rear, but firm and shell-like towards its head, has a more complex, definite bodily structure, with a strong hue, and is several times the size of any of the other entities
- Its proboscis form large, protruding claws at either side of its head, and it has a line of smaller, tooth-like proboscis folding inwards towards its mouth
- The venom it spits from its mouth is a bright green viscous liquid, which loses its opacity, speed and damaging power quickly, and glows in the dark
- It has large pigments covering much of its skin, particularly on its claws and the hard areas towards its front, and contains several (5 or more) glowing cores, which may appear to belong to several different individuals moving together in the dark

SECTION IV – USER INTERFACE

1 - Input System

- Because Lumens is aimed at a variety of devices (and a variety of devices can use the web application), all of their input mechanisms must be considered
- Each must provide ways to execute the game's commands
  - Move
  - Aim
  - Attack
  - Special combat modes
  - Beam
  - Repel
  - Range
  - Pause
- Input for menu navigation is trivial, and will not be covered here
- All of the following configurations have been devised with the player's comfort, familiarity, handedness, and personal preference and choice in mind

1.1 - Keys and Pointer (Mouse and Keyboard)

- The game may be played using either the keyboard alone or a combination of the mouse and keyboard
- The “up”, “down”, “left”, and “right” arrow keys, and the equivalent “w”, “s”, “a”, and “d” keys are familiar and widely used configurations for movement in games, so these move the protagonist here - in addition, to better support left-handed players using the keyboard and mouse, the “i”, “k”, “j”, and “l” keys will also perform the same functions
- The mouse adjusts the aim
- To execute a normal attack, the space bar or left mouse button is pressed
- The three enhanced combat modes – Beam, Repel, and Range – are activated by pressing and holding down the “x”, “c”, or “v” key, or the “m”, “n”, or “b” key, respectively, for the desired duration
- The “p”, “return”, and “g” keys all toggle pause on and off

1.2 - Pointers (Touch-screen)

- With all supported multi-touch input systems, Lumens again uses a combination of familiar input configurations and strategies addressing the other concerns
- For the in-game controls (i.e.: not the menu system), touch input is not limited to any particular zone of the screen
  - The player touches the screen, and further movements are measured relative to this initial point of contact
  - The order touches occur in is what decides which touch performs which in-game action
    - The first touch controls movement

        - The initial point of contact is encircled by a semi-transparent radius rendered on the screen, overlaid on the game so that it is clearly visible, but does not obstruct the image rendered below it
        - This circle represents the maximum applicable input, and denotes a zone on the screen, which affects what effect other touches have
    - The second touch outside the first zone controls aim and attacks
        - The initial point of contact is again encircled
        - If the touch hits or exceeds the edge of the circle, a normal attack in the direction the protagonist is aiming is executed continually until the touch ends or returns to within the circle
    - If at any point a double-tap occurs inside the two zones, a special attack is executed, for the duration of the touch
        - A double-tap inside the first (movement) zone activates Repel mode
        - A double-tap inside the second (aiming) zone activates Range mode
        - Two double-taps inside both zones (near-)simultaneously activates Beam mode
            - Since there is no way to fire the protagonist's proboscis in this mode, two simultaneous touches may be easily used for this mode
    - This system achieves several goals
        - The input controls do not get in the way of what the player wants to see, since they may touch the screen almost anywhere they wish
        - It supports handedness and other issues particular to the player in question through its flexibility
        - It requires a maximum of two simultaneous touches to execute all functionality
        - It should be very intuitive, flexible, and forgiving
- The game may be paused by touching the protagonist

1.3 - Buttons (Gamepad)

- The touch-screen system is inspired by gamepads with dual analogue sticks – just made more flexible to take advantage of the medium – and so these devices use a similar system
    - The left analogue stick controls movement; the right, aiming
    - These may be swapped in the options menu for left-handed players
    - The shoulder buttons (those controlled by the player's index and/or middle fingers) control combat on devices with 4
        - The top-right executes an attack
        - The top-left activates Range mode
        - The bottom-right activates Beam mode
        - The bottom-left activates Repel mode
    - The face buttons (the four on the right, pressed by the thumb) and directional buttons (the four on the left) provide fallbacks with some of the same functionality
        - The top face button activates Repel mode

        - The bottom face button activates Beam mode
        - The “up” directional button moves the protagonist forwards along its aiming line
        - The “down” directional button moves the protagonist backwards
        - The left and right face buttons, and the “left” and “right” directional buttons adjust the protagonist's aim anticlockwise and clockwise, respectively
        - This is useful both for catering to a player's preference, and for devices with fewer buttons
            - On those with only one analogue, the aiming may be done with the face buttons
            - For those with no analogue sticks, both aiming and movement can still be performed with these fallbacks
            - On devices with only 2 shoulder buttons, the Range and attack commands remain in place, but Repel and Beam are only activated by the face buttons
    - The usual pause button for the platform in question is used to pause the game – the “start” button on a PlayStation 3 controller, for example

1.4 - Other

- Other devices – such as the Wii controller and the XBox Kinect – may also be supported in the future

2 - Visual System

2.1 - Viewport

- The viewport is the visible area of the game environment – the environment's area may be larger than this, but it is not rendered to the screen outside of it
- It displays the game, menus, and information to the player
- It is elastically bound to the protagonist's position, and collides with the edges of the environment
    - As the protagonist moves, it exerts a force on the viewport to pull it along with it, subject to a maximum distance constraint between them which cannot be exceeded
    - If the edges of the environment meet those of the viewport, they collide, not allowing it to travel any further than the edge of the playable area

2.2 - Menus and Heads-Up-Display

- Menus and the HUD take up minimal visual space during the game
- Menus appear overlaid on the viewport
    - They have a semi-transparent white foggy background, covering the area around the text, blurring the view beneath, but not entirely obscuring it
    - The text has a coloured glass effect, the light of the protagonist making it glow a strong and almost opaque colour, and casting some subtle crepuscular rays through it

        - Its colour changes over time between colours that contrast strongly with the background fog
        - For the sake of legibility (and trying not to be too ridiculous), smaller text may omit some of these effects
    - These menus arrive and are dismissed like condensation appearing on glass when it's breathed on, then fading (though quickly), and are accompanied by a sound
    - There are some functions that appear on every menu screen
        - Exit: leave the game
        - Paused: un-pause the game, displayed in the upper-left corner
        - Back: return to the previous screen (un-pauses the game for the root screen)
        - Score: the current score and multiplier, if any, are displayed in the upper-right corner
- The HUD should be minimal or not required at all in this game, and is mentioned here to emphasise that idea
    - The following are the statuses which need to be displayed
        - The protagonist's health
            - The colour of the protagonist's center (the glowing point from which its light originates) and its aiming line proceed from gold to dull orange as it sustains damage
            - Its cast light follows, though always with a much higher lightness value (whiter)
        - The amount its enhanced combat modes have charged up
            - The three coloured glowing cores – one corresponding to each mode – are within its body
            - They fade when the corresponding mode has been depleted, and grow brighter as it recovers
        - The score and current multiplier
            - These must be displayed in order to be accurately represented, but are shown as innocuous menu-style text in the upper-right corner
            - However, the status of the multiplier is quite well represented by the auditory cues which accompany it, and both are shown in the pause menu and at the end of each stage, so the player has the option to turn their display off in the options menu, and go by these cues alone

2.3 - Screen Sizes and Orientation Changes

- Because of the variety of platforms that this application is aimed for (and could conceivably be run by), a fluid layout model is employed to ensure that it displays well on a large range of screen sizes and aspect ratios
    - A fluid layout is platform agnostic, and uses information at run time to decide on ideal sizes, to avoid aiming specifically at each platform and losing generality
    - Ideally, all sizing information is expressed as direct or indirect percentages of the basic screen size information available on each platform

        - This avoids targeting each platform individually, improves the application's ability to adapt well into the future, its flexibility, and its ability to handle resizing events gracefully
        - These changes will largely affect the menus and scale of the game environment
    - *Lumens* will respond to orientation changes on platforms which support them, or the more refined accelerometer events, where available
        - The viewport will simply rotate to the new orientation, with the menus following – for this reason, menu bounding boxes will be square or circular
        - The game will also re-orientate, with the aim of giving the impression that the screen and viewport are a rotating window into a stationary environment below

3 - Audio and Music

- Dynamic audio will be used throughout the game
    - Each significant event will have a base sound attached to it
    - This sound may be modified and altered on the fly according to the particulars of the event
        - For example, entities emit a sound relating to their motion
        - The base sound is altered in proportion to the entity's linear and rotational speed and acceleration, growing in volume
    - The base sounds and alterations are carefully selected to be musically complimentary
        - This requires much experimentation
        - Sounds may be synthetic, instrument samples, or other types which suit the themes of the game
- A metronomic beat runs in the background, forming the structure of the audio
    - All sound will enter a pipeline, which is delayed until a beat is reached and it can be played
    - This keeps everything in time, and helps prevent the sounds turning into an unmusical cacophony
    - The metronome describes and underlines this tempo
    - Very significant events – advancement through a stage, numbers of predators getting very close to the protagonist – may affect the overall tempo, to create tension
- The audio pipeline has other controls to ensure the sound does not become too loud or chaotic
- The overall effect is generated audio which is unique, descriptive of the events, interesting, and musical

SECTION V – ARTIFICIAL INTELLIGENCE

Entities use simple finite state machines and behaviour trees to govern the main parts of their behaviour.

1 - Antagonists

- Antagonists are all influenced by a variety of forces which combine to direct their behaviour, each weighted by their statistics, or handled by explicit programming
- The following behaviours are common to all predators (to differing degrees), and divided into the appropriate states
- Global state (behaviours always observed)
  - Wandering: movements which result in an indirect path being followed
  - Collision-avoidance: steer away from collisions with obstacles before they occur
- Passive state (while unlit)
  - The swarming behaviours (Reynolds, 2001)
    - Separation: avoid crowding neighbours
    - Alignment: steer towards the average heading of neighbours
    - Cohesion: stay close to neighbours
- Aggressive state (while lit)
  - Pursuit: largely overriding other behaviours for most predators, they chase down the protagonist
  - Evasion: move away from the aiming beam when lit by it, change direction and speed up if hit
  - Anticipation: use the protagonist's current velocity to aim towards the point it will be at if it continues its current movement
  - Attack: when in range, begin attacking – this may involve more complex behaviours in and of itself

1.1 - Fly

- The fly's behaviour is simply governed by the above influences – it has no other special behaviours

1.2 - Hunter

- To achieve its more active scouting and hunting behaviour, the Hunter observes some additional guidelines
- Passive state
  - Searching for prey: scout out areas the protagonist is likely to be in – the likelihood of their presence is weighted by some simple observations
    - The protagonist is not likely to be where neighbours are already looking, so search some distance away from the swarm

        - The protagonist may use obstacles for cover, and if they are near an obstacle, it is easier to corner them there than in open ground, so follow the edges of obstacles
        - When covering open areas, use a zigzagging, sweeping motion to cover ground better
        - If alone, they Hunter is vulnerable and less likely to corner their prey, and if the protagonist is between it and its neighbours, it should try to force it into a "pincer" trap – so return towards the swarm if it is too far away
    - Aggressive state
        - Advanced evasion: anticipate the aiming beam, moving away from it earlier
        - Stay away: keep to the edges of attacking range, as it requires more accuracy for the protagonist to strike successfully
        - Occasionally change between the following attacking strategies
            - Strafing: pick up speed, then drift and turn to face the protagonist while attacking, and repeat when speed reduces too much – lets momentum keep it a moving target while it attacks
            - Feint: make rapid changes in direction and speed
            - Rush: suddenly speed towards the protagonist while attacking, moving to "just miss" running into it, and end up on the other side of it – very hard to hit if it's moving quickly enough and catches the player off-guard, and lethal in groups

1.3 - Strobe

- Passive state
    - Strobing: occasionally have glowing cores fade, reducing influence from and on neighbours to zero with visibility – may disappear when vanishing control is "charged", must reappear when it "empties" and recover influence, but can choose any time in between those points at random – and while invisible, occasionally move in a wildly different direction at top speed before reappearing
- Aggressive state
    - Recast light: stay to the edges of the protagonist's cast light, so that the recast light travels as far an extra distance as possible, and keep moving with the rest of the swarm to try and attract them – only when being attacked, without nearby neighbours, or on infrequent occasions will it directly attack the protagonist, and it then immediately retreats by the fastest route to the edge

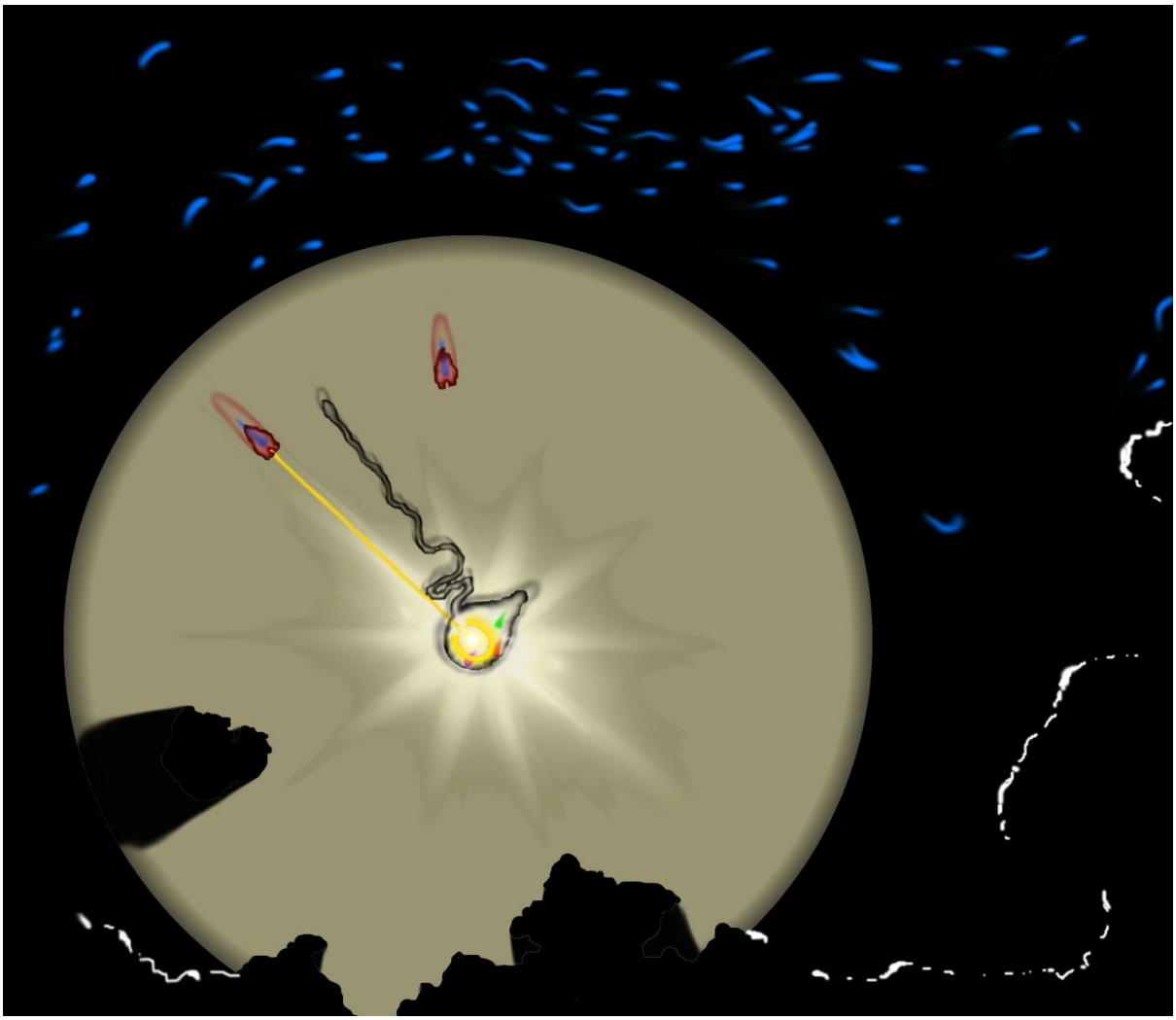
1.4 - Behemoth

- Aggressive state
    - Venom spit: if the protagonist is out of range, or on occasion, spit venom at it
    - Cannibalise: if severely damaged, turn towards a nearby neighbour (using anticipation to judge the best candidate) and pull it (with proboscis) to feed

2 - Protagonist

- While wandering, the protagonist observes the wandering and obstacle avoidance behaviours
- Pending play-testing, the protagonist may use an aim assisting behaviour

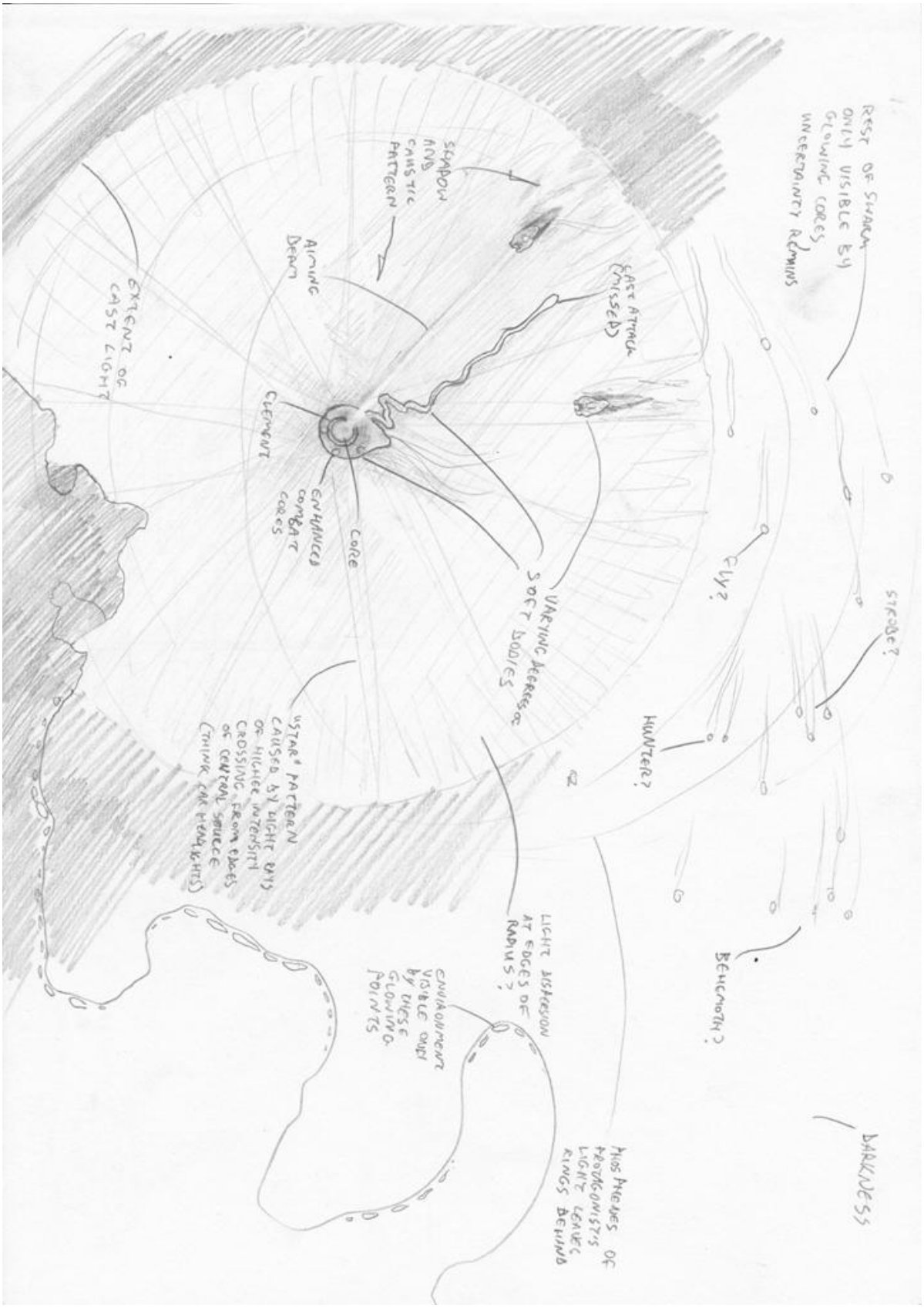
SECTION VI – CONCEPT ART



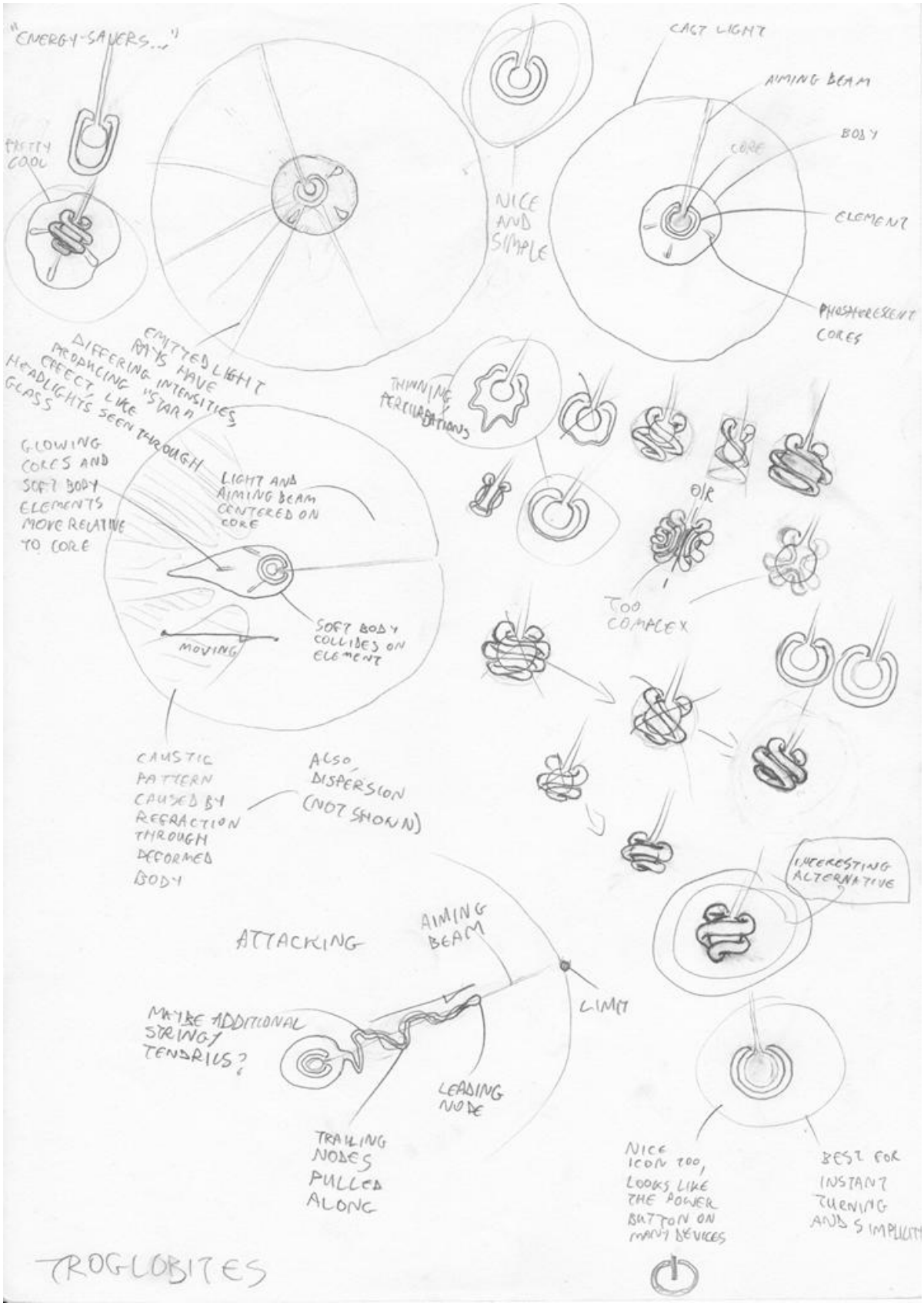
Mock-up of a typical game screen

Includes shadow-casting, caustics, soft-body physics, and terrain

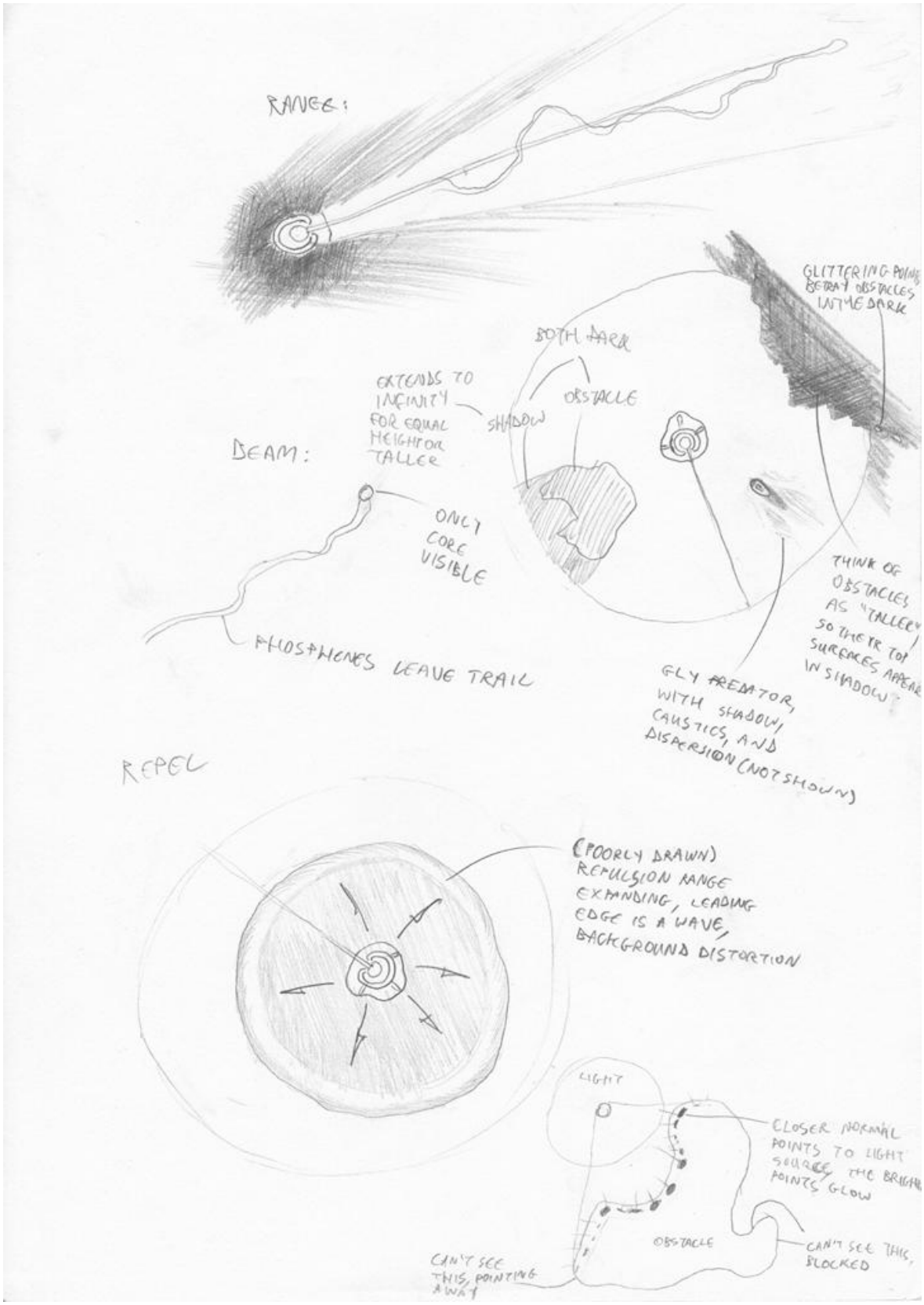
The swarm in the dark is depicted by their blue glows, and the edges of the terrain can be made out by their white glows



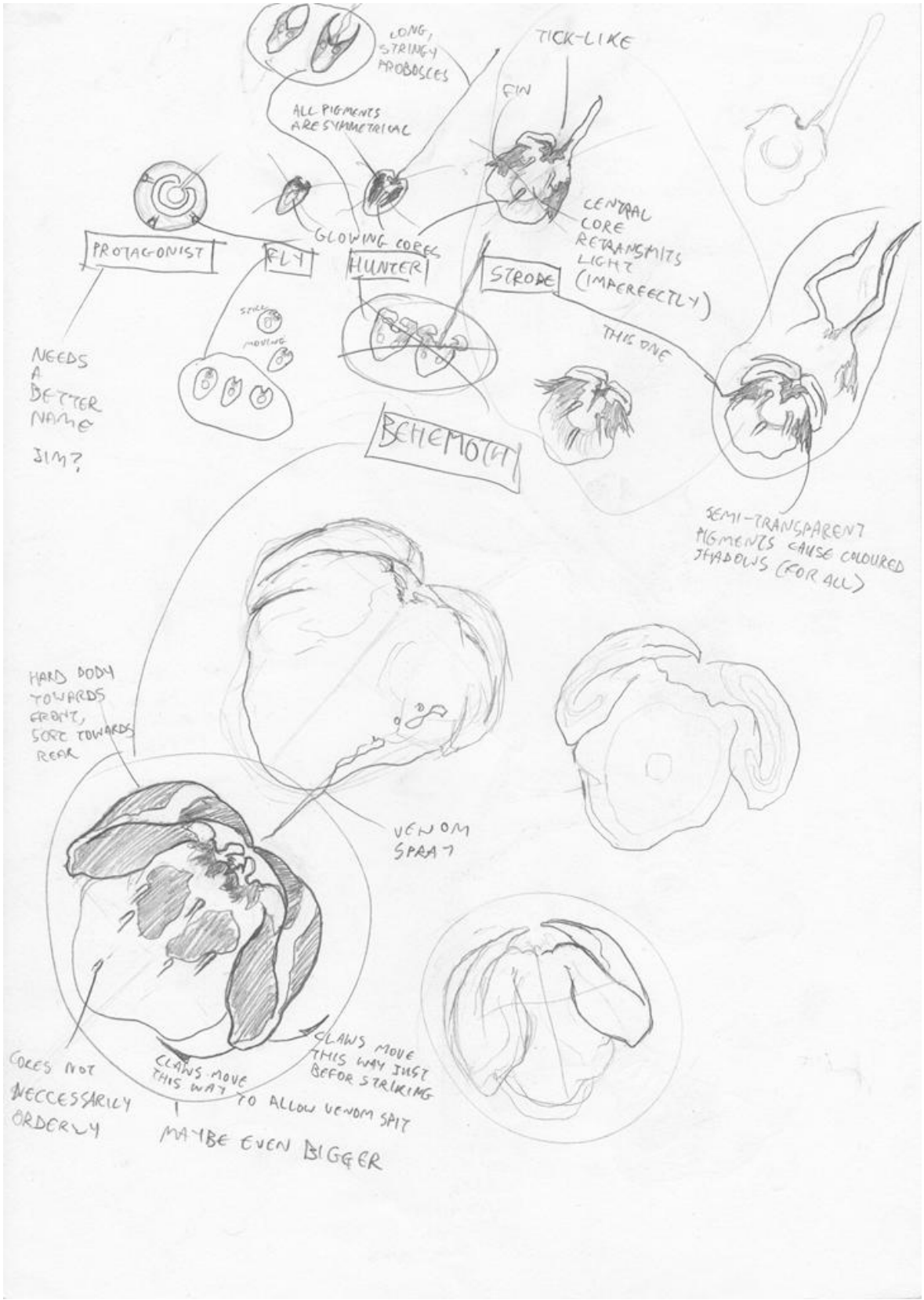
An overall view of a typical game screen



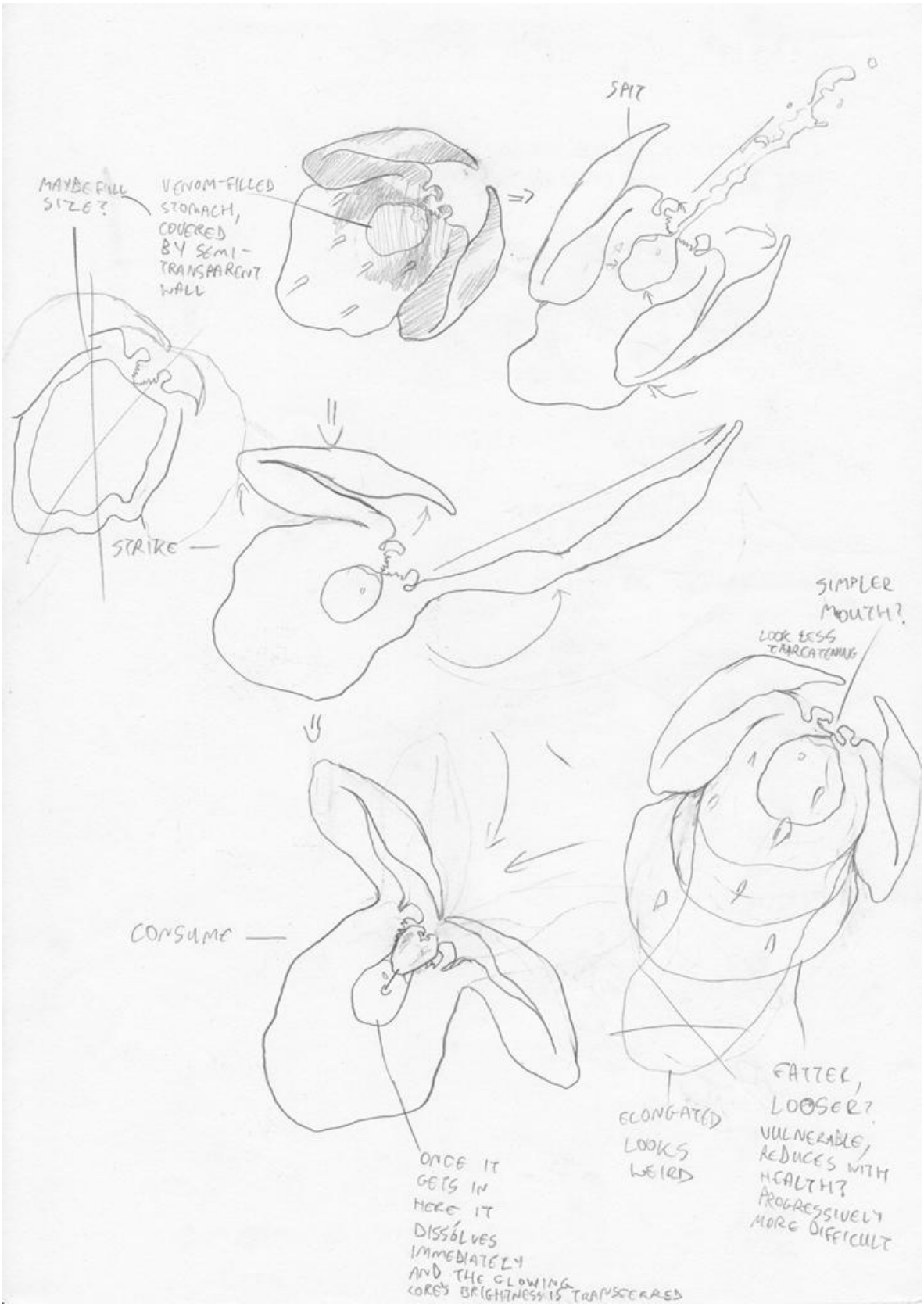
The protagonist's details



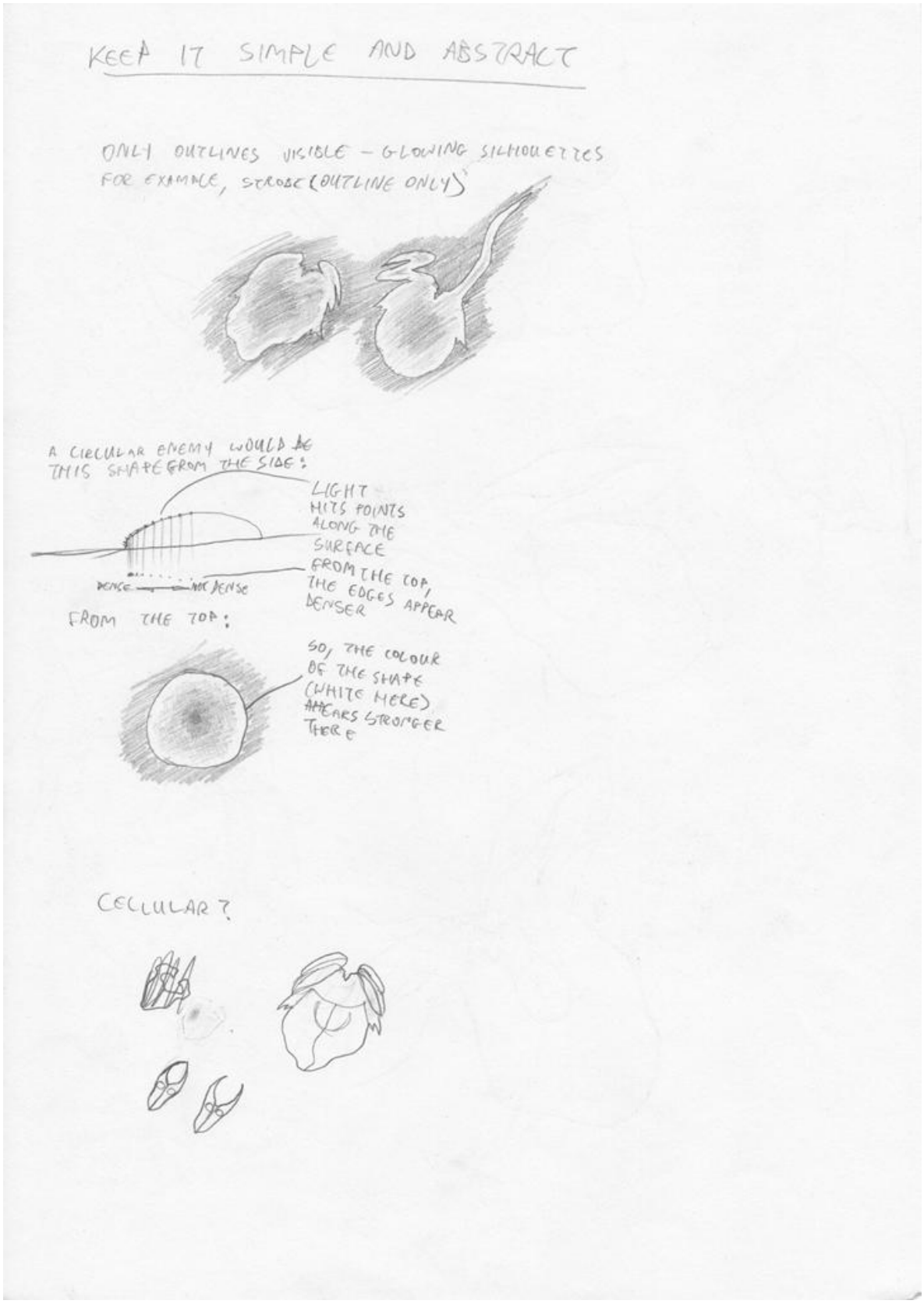
Enhanced combat and other details



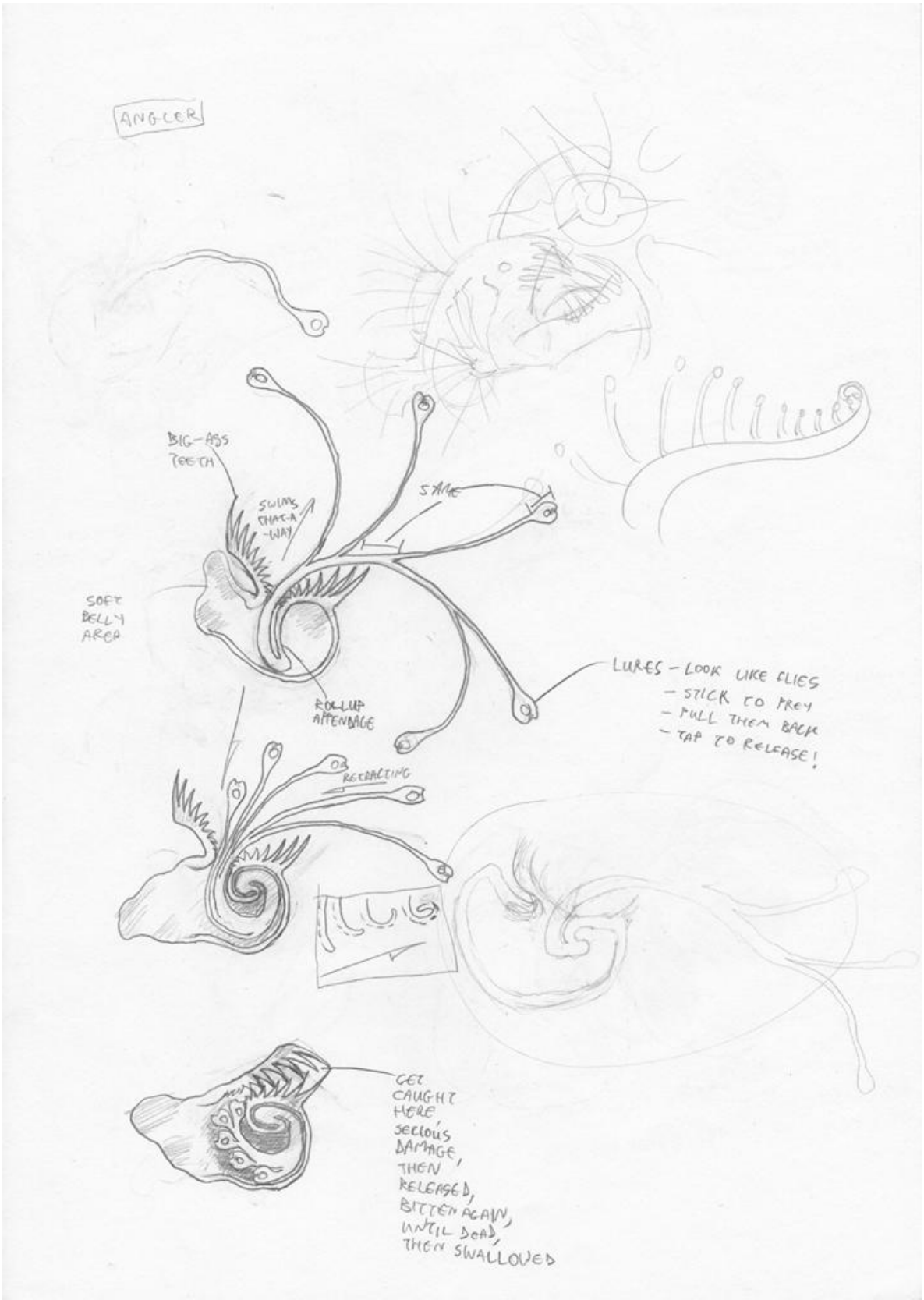
Predators and relative scales



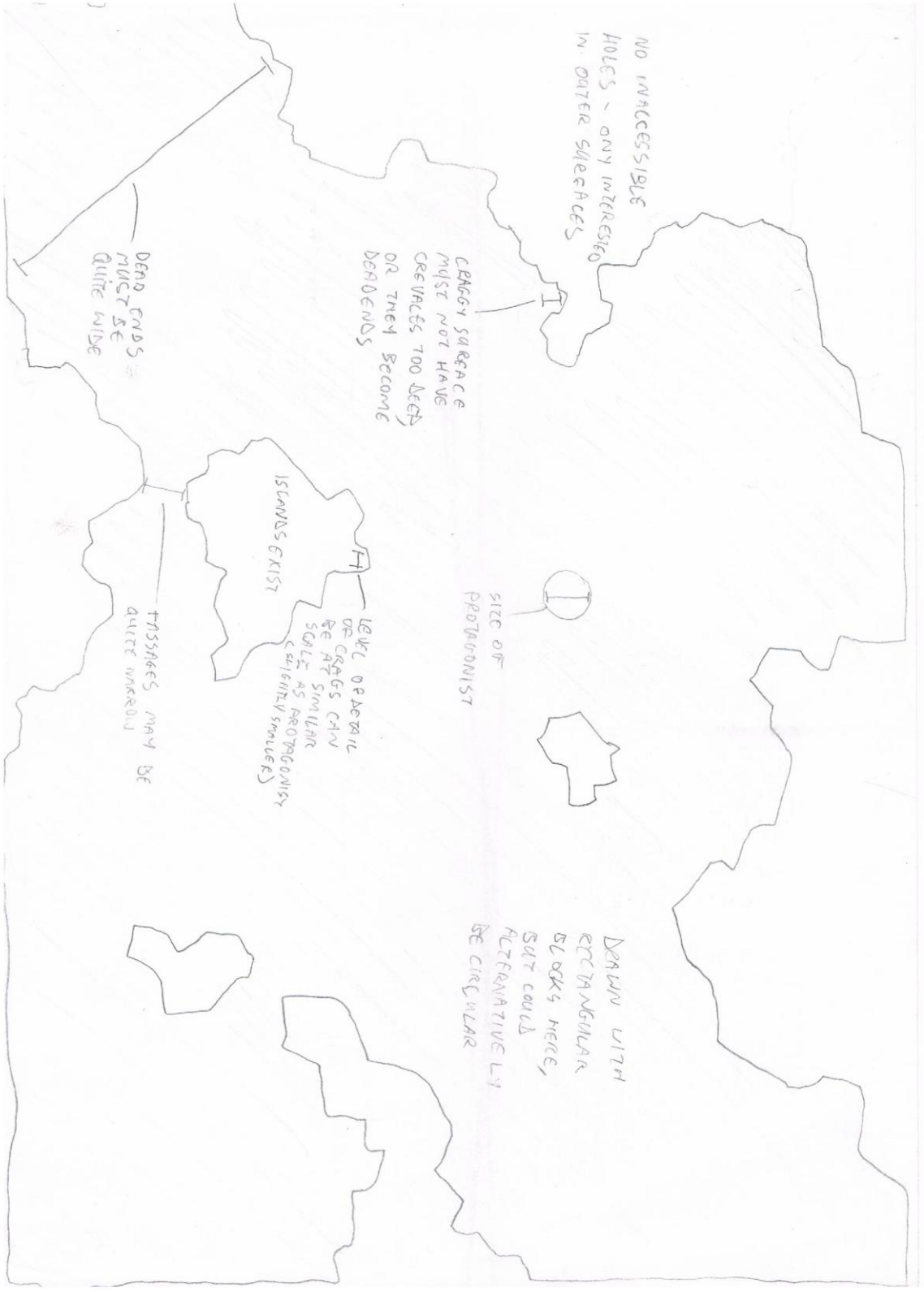
Behemoth variations and attacks



General entity look and feel



An additional predator type, for further expansion – the Angler



Level generation constraints sketch

SECTION VII –TECHNICAL

1 - Target Platforms and Considerations

1.1 - Modern Web Application

- The initial target, this will utilise recently emerging web technologies – WebGL, improved JavaScript engines, local storage, and other HTML5 innovations
- Lumens will be developed to the W3C specifications, and will not be not expected to run on browsers which do not correctly implement them, nor will great effort be spent on trying to make it work across a wide range of current and older browsers – the focus is on modern browsers
- It should, however, support a wide range of modern platforms which may access it, including mobile and touch-screen devices
- Speed and performance limitations are a major concern, since this is relatively new territory, and the project contains several computationally heavy components – flocking, real-time light simulation, dynamic audio, and other physics
    - Solutions to optimise performance on this and the other platforms will be discussed in further documents

1.2 - iOS

- The ideal solution would be to wrap an internal web browser context in a native application, with a communication interface between them to support persistence
- If this affects performance too severely, the context does not have the necessary features, or would violate the App Store Guidelines, then a full port may be required
- The App Store Guidelines and quality control measures must be closely adhered to
- Performance is an issue of even greater significance on mobile platforms

1.3 - PSP

- This platform will be aimed for as an attempt to make use of the access to development kits I have during the course of this project
- It will require a full port of the game, and very likely be a great challenge

1.4 - Android

- The considerations for this platform are similar to those for iOS, with the exception of the App Store Guidelines

1.5 - Windows

- A native Windows application would reduce the constraints on performance
- Implementing the existing web application (if it is successful) through a browser context may be a preferable option in terms of development time

1.6 - Mac OS

- A native Mac application has the same considerations as Windows, with the addition of the Mac App Store Guidelines

2 - Development Environment

2.1 - Programming Languages

- JavaScript (with jQuery, Modernizr and Three.js), HTML (HTML5), CSS (CSS3, SCSS) – Web Application, fast prototyping
- GLSL – Shaders
- C, C++ – iOS, Mac, Windows, PSP
- Objective C – iOS, Mac
- Java – Android
- XML (or other platform-independent markup) – communication and persistence

2.2 - Software

- Sublime Text 2
- Chrome developer tools (and Webkit, Firebug, and other in-browser development tools)
- Visual Studio 2010/2008
- XCode 4
- GitHub
- Photoshop (and other asset-production software)
- WAMP Server
- Frameworks and libraries
  - jQuery/Prototype
  - Modernizr.js
  - Three.js
  - Node.js
  - Sony PSP development plugins for Visual Studio 2008
  - Cocos2D/Box2D

2.3 - Hardware

- Dell Studio 1558, Intel Q740 Core i7, 4GB RAM, Windows 7 Home Premium/Professional
- PSP development kit
- MacBook Pro

SECTION VIII – INITIAL PROJECT MANAGEMENT AND OBJECTIVES

1 - Approach

- The project will be developed using an adaptive, iterative approach
- Working versions will be produced, incrementally adding features
- Adaptations will be made as the need arises and as reviews and reflections are carried out at each iteration
- The project may change substantially, in keeping with its general aims and themes

2 - Iterations

2.1 - Game-play Mechanisms and Basic Physics

- Iteration 0.1 will be a demonstration of the game-play mechanics
  - Protagonist, with movement, health, and normal combat
  - Fly predator swarm, with full behaviours
  - Full core negative feedback loop in place
  - Initial game balancing
- It will include the minimum necessary physics and visual rendering required for this purpose
- The central modules will be established, and a modular application architecture will be in place for other components – rendering, user input, etc.
- This is the core of the project

2.2 - Basic Light Simulation

- Iteration 0.2 will begin exploring the basics of light simulation
  - Lighting
  - Shadow casting
  - Non-ray-traced phosphorescence
- Good performance and a scalable architecture must also be achieved
- This is the first phase of the real-time light simulation challenge

2.3 - Advanced Light Simulation

- Iteration 0.3 will achieve the advanced goals of the light simulation
  - Reflection, with bleeding colours
  - Refraction, with dispersion
  - Caustics, with sharp patterns
  - After-image
  - Crepuscular rays and volumetric light
  - Rays of varying intensity
  - Reaching a good approximate solution for the rendering equation
- Performance optimisations must be implemented
- This is the final phase of the real-time light simulation challenge

2.4 - Advanced Physics

- Iteration 0.4 will have in place a physics engine comprehensively covering the use-cases required in the game
  - Particles
  - Soft bodies, both “open” and “closed”
  - Rigid bodies
  - Collision detection and resolution engine
  - Approximated air-flow reaction for atmospheric particles
  - Magnetic attraction for collectibles
- This will be achieved in a modular, scalable physics engine
- This, along with flocking and lighting, is the third major technical milestone

2.5 - Further Variety in Game-play and Antagonists

- At the stage of Iteration 0.5, the other antagonists will all be fully integrated into the game
- The combat systems and artificial intelligence for the protagonist and antagonists will be in place
- Scoring and multipliers will largely be in place, recording and balanced
- The game-play and visual elements will have reached a fully working stage representative of the eventual release

2.6 - Dynamic Audio

- Full dynamic audio will arrive in Iteration 0.6
  - All audible events will produce sound and alter it suitably with variables
  - Sound pipeline/channel will be in place and managing all incoming requests
- This finishes out the core aesthetic elements of the game

2.7 - Stage Generation and Progression

- Iteration 0.7 will move the game from a demonstration mode to introduce advancement
- The stage generation module should generate reliable results for arbitrary arguments, and therefore be capable of producing a near-infinite variety of distinct stages
- Persistence will be introduced at this stage in relation to stage generation and retrieval

2.8 - Menu System

- Iteration 0.8 will adorn the game with the necessary trimmings for users to manage it, and support local user profiles
- Persistence will be completed to this local level

2.9 - Scores

- Iteration 0.9 will fully support user profiles
- Recording and retrieving scores, finishing off persistence

- Full necessary communications for the first target platform to manage user profiles via a server

2.10 - Tightening Up and First Platform Launch

- Any remaining major bugs will be removed, tweaks made, and a final rebalancing will be done before Iteration 1.0 is released for testing
- Pending testing and rebalancing, *Lumens* will be launched on its first platform, possibly commercially

2.11 - Revisions, Changes and Subsequent Platform Launches

- Iterations X.X will repeat as many of the above steps as necessary to secure releases on some or all of the target platforms
- Changes and enhancements are expected to take place during these stages
- The application may also require a commercial strategy, if the results are marketable, which will be explored later

PART II – IMPLEMENTATION CHALLENGES AND PROCESS

OVERVIEW

In the design phase, *Section VIII: 2*, targets for the iterative development of the project were laid out, which provided an idea of the priorities of the project – timelines were estimated after this as well, but its main purpose was to provide a general structure. The adaptive spirit of the development process would override these as more was learned during development, necessitating changes as some components would turn out to be dependent on others.

This adaptive methodology became important almost immediately during the development of the second prototype – iteration 0.1, developed under the revised system design with an improved approach to physics and architecture over the first prototype.
After setting up the framework of the project and achieving the basic milestones of that iteration – basic physics and rendering, predators with full swarming behaviours – it appeared that the implementation of the basic game-play should be postponed until basic shadow-casting was in place.

There were two reasons for this decision.

Real-time ray-tracing was the most interesting and challenging aspect of the project – after the basic skeleton was in place – and I wanted to tackle it immediately, in order to ensure it was addressed in some way before the final deadline. I would prefer to have faced some new challenge, learned something, and achieved something demonstrable from it; rather than have spent more time developing the same kind of thing I have before and discovering too late that the greater challenges required more time than I had remaining to overcome, leaving me with something I could have put together before learning from this project.
It was a simple choice of addressing the biggest challenge first. The other components could be developed any time and so taken for granted for the time being, but the most difficult problems needed to be tackled first.
The opposing argument would be that introducing a basic working version of the game first and addressing the more complex issues later would have made it easier to communicate and demonstrate the idea, as well as test it; but I could already feel the time passing, and was eager to cut to the core issue as quickly as possible – I decided to get stuck straight into the interesting part.

The finished game would require accurate detection to check whether or not a predator is lit. This was a core aspect of the game – the player would not otherwise be able to use the environment as cover, which is absolutely necessary.
The system developed for a basic working prototype would not represent the finished article well – since it would not account for this important mechanic – and would also take more time to implement than I wished to spend – only to be largely replaced again later. In order to better represent the game-play flow as it would eventually be, I needed to use the GPU ray-tracing system to return this information.
Again, there is the opposing argument that even a simple prototype, not fully representative of the final product, would still serve to provide a sketch of the idea – but, in this particular case, I felt this component would be too important to leave out on a small time budget.

This was a key decision in changing the course of the project, but I stand by it, as I learned far more along this path than I would have from another. If this project was being developed for a client eager to see results early on, things would probably have been different; but I had a limited amount of time to get the most from this project for myself, and did so.

The overall course of development resulted in successful demonstration of the key motion and style of the game, a simplified shadow-casting system, and a more advanced system ready for further expansion on a more capable platform.
It achieved the goals laid out up to iteration 0.2 – without the full game-play prototype, as discussed above –including modular, extensible game architecture as designed; emergent behaviours, with up to 600 entities at one time running at interactive speeds; and sphere-based shadow-casting, developed from several sources and including a system for the dynamic passing of global data from CPU to GPU. There is also a more advanced ray-tracing system as per iteration 0.3 developed, though too heavy for the platform in use.
At the outset of development, I had hoped to reach iteration 0.4 or 0.5, but those speculated targets did not account for some of the unforeseen implementation difficulties encountered; so, considering this, I am reasonably content about the work achieved.

The following sections deal with the major areas tackled over the course of development and the process used; challenges encountered, research done, efforts made, and solutions arrived at; sample source code and detail on key points of interest; and thoughts, reflections, things learned and changes I would make were I to do it all again.

The full source code is available from GitHub (https://github.com/keeffEoghan/Lumens).
Of particular interest are the head revision – which implements shadow-casting – and the revision before it, which demonstrates the entities running at full potential, without ray-tracing (https://github.com/keeffEoghan/Lumens/commit/b46583efe14d4f5eab539d9b4a072345257bf605).
The commit messages have also acted as a log of activity since I started using GitHub, so they may themselves be of interest.

SECTION I – FRAMEWORK

1 - Core features

1.1 - Space-Partitioning System

It was evident early in the design process – after initial research into the emergent flocking behaviours (Reynolds, 2001), ray-tracing, and collision-detection – that a space-partitioning data structure would be vital to accelerating these operations. Such a system groups collections of entities together according to their positions in space, allowing entities that are near a certain position in space to be queried and retrieved without iterating over every entity and checking its position. This is a performance optimisation, but needed to be considered at the start of development, since so many things are based on it, and it would be more difficult to crowbar it into the system later on.

A quad-tree system seemed to be the most appropriate for the project (since most computation was done in two dimensions and the rest could be handled simply, an octree system was not required), and so I settled on an MIT-licensed JavaScript implementation (Chambers, 2012). This system was adapted further in order to better fit into the project, and to provide more functionality.

A core addition was a two-way system. The original implementation was designed for entities to only be added to the tree, and not removed or updated – instead, the entire tree was cleared and rebuilt with the updated entities at the start of every step. With careful ordering of operations, this system worked fine:

- Clear the tree
- Resolve each entity's new position
- Place it in the tree
- Carry out operations which rely on the quad-tree
- Accumulate updates for the next step

That was, until the collision detection and resolution system was being developed, which both depended on the quad-tree and updated its entities' positions, and so did not fit into this pattern. Therefore, a two-way system was added, which allowed entities to be removed and replaced without clearing the entire tree.

This was achieved by altering the clear function to accept individual and arrays of entities to be cleared, and to have the node check its sub-nodes to see if their entities could be merged back up the tree to itself; and by adding a new merge function to facilitate this.

```
…
clear: function(item) {
   if(item) {
      // Note: item here refers to the exact object stored in the
tree
      if($.isArray(item)) {
        for(var i = 0; i < item.length; ++i) { this.clear(item[i]); }
      }
      else if(this.nodes.length) {
```

```
         var node = this.nodesFor(item)[0];

         if(node) {
            if(node.nodes.length) { node.clear(item); }
            else if(node.kids.remove(item)) {
               var nodeKids = 0;

               for(var l = 0; l < this.nodes.length; ++l) {
                  var nL = this.nodes[l];

                  // Don't merge if node still has subnodes
                  if(nL.nodes.length) {
                     nodeKids = Infinity;
                     break;
                  }
                  else { nodeKids += nL.kids.length; }
               }

               if(nodeKids <= this.maxKids) { this.merge(); }
            }
            else {
               //log("QuadTree Warning: item to be cleared not …”);
            }
         }
      }
      else if(!this.kids.remove(item)) {
         //log("QuadTree Warning: item to be cleared not found");
      }
   }
   else {
      this.kids.length = 0;

      for(var n = 0; n < this.nodes.length; ++n) {
         this.nodes[n].clear();
      }

      this.nodes.length = 0;
   }

   // For function chaining
   return this;
},
…
merge: function() {
   for(var n = 0; n < this.nodes.length; ++n) {
      var node = this.nodes[n].merge();

      this.kids = this.kids.concat(node.kids);
      node.clear();
   }
```

```
    this.nodes.length = 0;

    return this;
}
…
```

This ensured the desired number of entities was maintained in each node, and the quad-tree structure was adhered to; without containing duplicates, empty nodes, or other issues.

There are a few alternatives to quad-trees, probably the most suitable of them being the KD-Tree. Its method of partitioning space is more sophisticated than the quad-tree's, so it may have proved a slightly more efficient system to use for this project; however, the quad-tree implementation was quite effective, simple, and served the purpose well. So I feel happy with it until the project reaches the final optimisation stages, where the boost of a KD-Tree could be worth investing time in – and the interface should remain the same, so it should slot in quite neatly at that stage.

2 - Challenges

2.1 - Design and Structure

Overall, structuring the project was not a major issue. I found taking some time to think about and design it with tools like UML diagrams, or simply a pencil and paper, allowed me to clarify the direction I was going in – and avoid the tendency to wander aimlessly and miss the bigger picture.

2.2 - JavaScript

I did find some issues programming a larger class structure in JavaScript, however. It is not a fully object-oriented language, but rather a prototypical one (one without classes); and I found that getting used to some familiar patterns – such as inheritance or template classes – in JavaScript was not always straightforward.

This sample code, for example, shows a function defined to encapsulate prototypical inheritance in a more convenient way:

```
function inherit(Child, Parent) {
    Child.prototype = Object.create(Parent.prototype);
    Child.prototype.constructor = Child;

    return Child;
}
```

Best-practice methods from several sources (Mozilla, Resig; 2012) were collected, and this function was defined to conveniently and safely handle general prototypical inheritance in one place.
As with almost all of the JavaScript code in *Lumens*, the function returns the most useful object, to facilitate function-call-chaining and further reduce the verbosity of code. For

example, a typical use would have a child class inherit from a parent, then extend the child's prototype (using jQuery's `$.extend` function) to add new functionality, or override the parent's functionality:

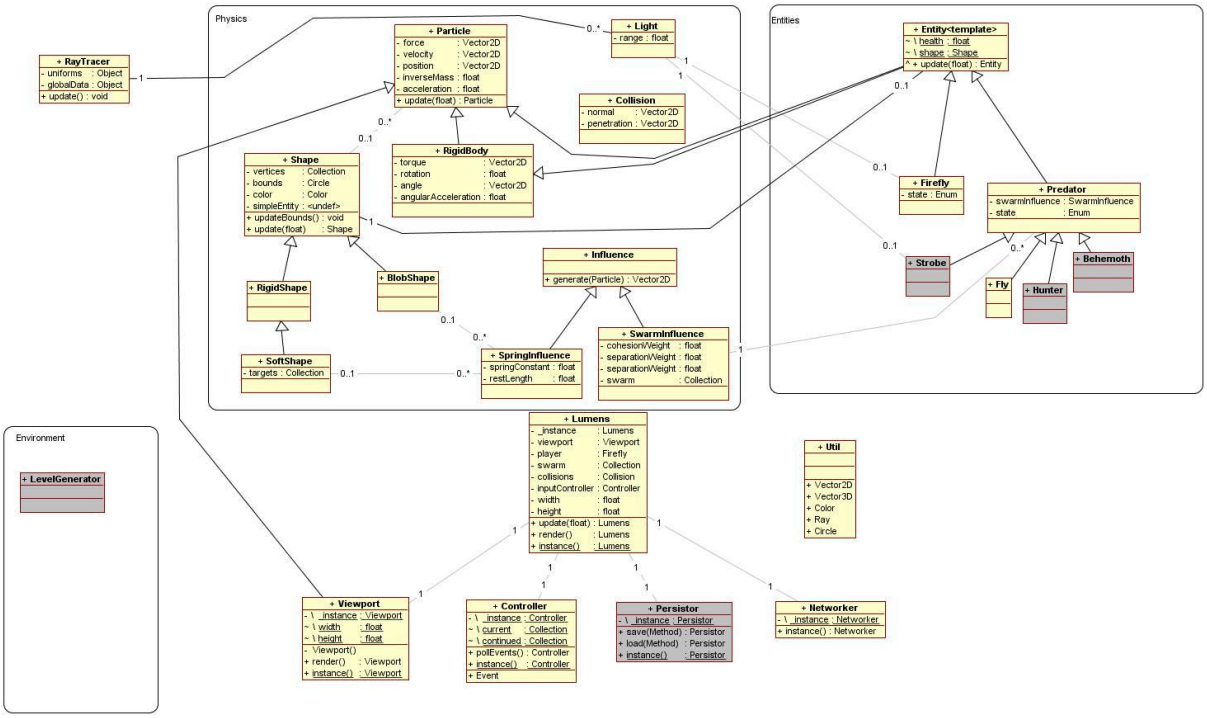
```
// Parent "class" and constructor
function Parent(…) {…}
// Functionality added to parent's prototype
$.extend(Parent.prototype, {…});

// Child class
function Child(…) {
   …
   // Call to super constructor
   Parent.call(this, …);
   …
}
// Inheritance and polymorphism (overridden functionality in the
subclass)
$.extend(inherit(Child, Parent).prototype, {…});
```

Seeing the differences between a stricter language and a dynamic one was an interesting aspect of the process of development.
JavaScript will let you do a lot more, code awful things, and get away with it for a time. But it will punish you later with some obscure and inscrutable issue, often with unhelpful or even misleading indications of what's gone wrong.
A stricter language, however, will only let you do things one way, and if you deviate from it, will complain until you fix it. But you are usually surer of where you stand with the program you've written.

Finding reliable references for JavaScript was very important, as for every right way to do something, there's several wrong ones being used and shared. The Mozilla Developer Network (2012), became an invaluable resource, along with various accomplished web developers (Cabello, Irish, Hattab, Lewis (1), Resig; 2012), who's work I often examined to guide my own.

3 - Results and Reflection

The project framework was developed as shown in the following diagram (with some parts excluded for brevity, and the greyed out classes indicating those which have not yet been implemented).



Some components were written according to the design and never implemented, such as the touch controller interface.

Overall – after some refactoring – it proved a solid structure, and was suitably constructed to allow additional elements to be introduced with relative ease.

SECTION II – PHYSICS

1 - Research and Implementation

The physics engine was developed to the point of particle physics, adapted from the physics engine design of a practical mathematics course delivered in WIT, and Ian Millington's *Game Physics Engine Development* (2007), its excellent primary source. Adapted from this material were Newtonian particle physics; force and impulse generators (springs and collision responses); and collision detection and resolution.
Some of these systems were adapted further from the existing engine, and others added, which we will examine in further detail – the rest is pretty close to the C++ implementation in the aforementioned reference material.

1.1 - Collision Detection

The collision detection and resolution system went through a couple of iterations; initially trying to more closely follow Millington's collective solving system.
This involved all collisions being detected and reported to a central handling system. This system would then iterate several times over all the collisions, resolving each pair and then rechecking them for collisions resulting from the first resolution – effectively "shuffling" multiple collisions apart over several steps, until the entire system had been resolved to a state of no collision.

It became apparent that this system would demand too many resources, impacting upon performance and development time, and was superfluous for the particular purposes of this project. So, it was then adapted into a simpler discrete system.
In this system, when a collision was detected, it was resolved immediately and over a single step for the two entities involved. Each entity was in turn updated and checked for collisions with its neighbours, so all entities were checked at least once.
This system forgoes some of the accuracy of Millington's method, favouring performance. But it is fit for purpose in this case, since the number of entities on screen at any one time is a higher priority than exact physical accuracy.

The system was developed to the point of bounds-checking – responding to an entity's bounding sphere, rather than its actual geometry – with the intention of continuing to geometry-checking if the ray-tracing system advanced sufficiently to make it worthwhile. Up until the ray-traced iteration, entity bounding radii were checked against the actual geometry of the environmental obstacles.

1.2 - Damped Springs

A simple adaptation of the standard spring force, with damping applied to oppose velocity in the direction of the end of the spring.

```
…
dampedSpring: function(options) {
```

```
   // The normal spring force takes a Vec2D, not a Particle
   var from = options.pointFrom.pos, oldFrom = options.posFrom;

   options.posFrom = from;

   //Apply the normal spring force
   var spring = this.spring(options), damp;

   if(from.equals(options.posTo)) {
      damp = options.pointFrom.vel.scale(-options.damping);
   }
   else {
      damp = from.sub(options.posTo).doUnit();
      damp.doScale(-options.damping*Math.max(0,
         options.pointFrom.vel.dot(damp)));
   }

   // Not using this any more
   delete options.posFrom;
   options.posFrom = oldFrom;

   return damp.doAdd(spring);
},
…
```

1.3 - Pressure Soft Bodies

The simplest approach to simulating a soft body is to create a mass aggregate model, constructed of particles attached by springs. This is liable to become unstable, however; and it can be difficult to get the balance correct between setting up enough springs in the right places for the shape to be stable, and not overburdening the simulation with too many (for example, from every particle in the model to every other particle).

So, after looking around for a better way, I read a paper about a pressure-based system (Matyka, 2004). Matyka describes the method as like having a gas-filled material, whose surface is composed of particles connected by spring, but which is kept inflated by the pressure of the gas within. Thus, each particle need only be connected to its direct neighbours (defining the shape's surface), and not have strut springs connected to opposite particles. The separating force is pressure instead – calculated once and applied equally to all particles – far more efficient than several spring forces.

In order to calculate the pressure force, the volume of the body is approximated using Gauss theorem – based on the surface of the body rather than its actual volume.

```
…
applyPressure: function(points, density) {
   // Derived used to prevent repeated calculations
   var volume = 0, derived = [];

   // Gauss theorem for approximating volume
   for(var p = 0; p < points.length; ++p) {
```

```
      var from = points.wrap(p-1), to = points[p],
         vec = to.pos.sub(from.pos),
         mag = vec.mag();

      if(mag) {
         var normal = vec.perp().doScale(1/mag);

         volume += 0.5*Math.abs(vec.x)*mag*Math.abs(normal.x);
         derived.push({ from: from, to: to, mag: mag, normal:
normal});
      }
   }

   var invVolume = 1/volume;
   for(var d = 0; d < derived.length; ++d) {
      // Apply pressure
      var data = derived[d], pressure = invVolume*density*data.mag;

      data.from.force.doAdd(data.normal.scale(pressure));
      data.to.force.doAdd(data.normal.scale(pressure));
   }
}
```

2 - Results and Reflection

The physics engine as a whole worked very well eventually. It's still quite a simple affair, without rigid body simulation, a more robust collision system, or geometry-level checking for most entities. However, it was able to simulate up to around 600 entities at a time (probably more if other features had been disabled) at interactive rates of 25-35 fps. Seeing hundreds of swarming entities flowing around the screen, colliding and interacting with the environment and one another in real time was quite rewarding.

However, it took a considerable amount of refactoring and small optimisations to get it running at that rate. The quad-tree system was invaluable, but there were many smaller changes that needed to be made, and considerations to be made for the way JavaScript works, for example:

- Caching variables (keeping direct references to properties of other objects used repeatedly) is important, since the process of resolving a query for a property can be expensive in JavaScript
- Guarding as many operations as possible with conditional breaks and early exit opportunities proved to save a lot in terms of performance

The process of researching and implementing the physics engine and related features was, for the most part, a very interesting and engaging one – seeing the clever processes devised by the authors of the resources I studied made for very good reading.

Though it was not used in the main prototypes in the end, just its own test, the pressure soft body simulation was particularly interesting.

Another enjoyable application of the engine was the viewport – attached to the protagonist by a damped spring, and colliding with the sides of the level area when small enough to fit within them, it resulted in quite a nice, smooth camera panning motion.

SECTION III – ARTIFICIAL INTELLIGENCE

1 - Research and Implementation

1.1 - Swarm

A core part of the concept of *Lumens* is the swarming behaviour of the predators. The algorithm is a very famous, and fascinatingly simple one – Craig Reynolds' "Boids" (2001). Having covered this ground before in the first prototype, a few things had become apparent:

- The first version was limited to fewer entities than I would like, and so the quad-tree system would be a necessity to bring the algorithm down from $O(n^2)$ to something which would scale better
- There was a very simple logical error in the first attempt, which produced some oddly predictable behaviour (this was overlooked for longer than I would like to admit)

The main point of note here is that the first prototype revealed the necessity of a space-partitioning data structure, which is why the project was redesigned with that in mind.

The algorithm itself was quite trivial to implement (Buckland, 2005; Brundage, 2011), with some optimisations to reduce the number of computations where possible, and other minor changes (such as calculating from a point some way towards the entity's future position:

```
…
swarm: function(options) {
   var totalSeparation = new Vec2D(), totalCohesion = new Vec2D(),
      totalAlignment = new Vec2D(), swarm = new Vec2D(),

      predict = (options.predict || 0),
      // Aim ahead of current position
      focus = options.member.vel.scale(predict)
         .doAdd(options.member.pos),
      // Query the QuadTree
      nearby = options.swarm.get((new Circle(focus,
            options.nearbyRad)).containingAARect()),

      num = 0;

   for(var n = 0; n < nearby.length; ++n) {
      var near = nearby[n].item;

      if(options.member !== near) {
         var nearbyFocus = near.vel.scale(predict)
               .doAdd(near.pos),
            vec = focus.sub(nearbyFocus),
            dist = vec.mag();

         if(dist < options.nearbyRad) {
            ++num;
```

```
            var distFactor = options.nearbyRad/dist;
            totalSeparation.doAdd(vec.doUnit()
               .doScale(distFactor*distFactor));

            totalCohesion.doAdd(nearbyFocus).doSub(focus);

            var alignment = (near.angle || ((near.vel.magSq())?
                  near.vel.unit() : null));

            if(alignment) { totalAlignment.doAdd(alignment); }
         }
      }
   }

   if(num) {
      // Calculate averages
      swarm.doAdd(totalSeparation.doScale(options.weight.separation))
         .doAdd(totalCohesion.doScale(options.weight.cohesion))
         .doAdd(totalAlignment.doScale(options.weight.alignment))
         .doScale(1/num);
   }

   return swarm;
},
…
```

1.2 - Wander

The wander force is a simple adaptation of Mat Buckland's (2005), in which a circle is projected ahead of the entity, and a random point along its circumference selected as a target position. This results in a smooth, fishlike meandering, which serves a few purposes:

- Keeping isolated entities moving
- Making their movements look more interesting
- Adding some instability to the swarm behaviour, allowing entities to break away and increasing unpredictability

```
…
wander: function(options) {
   /* range is proportional to the distance either side of the
current
      heading within which the entity may wander (radius of wander
circle)
      vel is the minimum velocity vector which wandering may produce
      (distance wander circle is ahead of the entity) */
   var angle = Math.random()*2*Math.PI;
   return options.vel.add(new Vec2D(options.range*Math.cos(angle),
      options.range*Math.sin(angle)));
},
…
```

1.3 - Obstacle Avoidance

Similarly to the wander and swarm behaviours, a circle of influence is projected some way towards the entity's next position.

This circle is checked for collisions with nearby environmental obstacles, and a force proportional to the square of the collision's penetration is applied along the normal of the collision, steering the entity away from the wall before it hits it, and growing exponentially the closer the entity is to the wall.

```
…
avoidWalls: function(options) {
   var force = new Vec2D(),
      predict = (options.predict || 0),
      focus = new Circle(options.point.vel.scale(predict)
            .doAdd(options.point.pos), options.radius),
      nearby = options.walls.get(focus.containingAARect());

   for(var w = 0; w < nearby.length; ++w) {
      var wall = nearby[w].item,
         shape = wall.shape;

      if(shape.boundRad.intersects(focus)) {
         var verts = shape.three.geometry.vertices;

         for(var v = 0; v < verts.length; ++v) {
            var a = shape.globPos(verts.wrap(v-1).position),
               b = shape.globPos(verts[v].position),
               c = Collision.Circle.checkLine(focus, a, b);

            if(c) {
               // Inverse square force (already divided by pen)
               force.doAdd(c.vector.scale(c.penetration));
            }
         }
      }
   }

   if(options.lumens) {
      var lc = Collision.Lumens.checkRect(options.lumens,
            focus.containingAARect());

      if(lc) {
         lc.doSwap();

         // Inverse square force
         force.doAdd(lc.vector.scale(lc.penetration));
      }
   }

   return force;
```

```
},
…
```

1.4 - Scheduling System

Because some of the AI operations are expensive – swarming and obstacle avoidance in particular – and not as time-sensitive as physics or other concerns, a scheduling system was put in place to control and reduce the number of times these operations would be called.

```
function Schedule(wait, last) {
   this.wait = (wait || 0);
   this.last = (last || 0);
}
$.extend(Schedule.prototype, {
   check: function(time) {
      return (time-this.last >= this.wait);
   },
   copy: function(other) {
      if(other) {
         this.wait = other.wait;
         this.last = other.last;
         return this;
      }
      else {
         return new this.constructor(this.wait, this.last);
      }
   }
});
…
   …
   // In Predator's update function
   if(this.swarm.schedule.check(time)) {
      // Contributes to the predator's guiding force
      this.swarm.force = Force.swarm(this.swarm);
      this.swarm.schedule.last = time;
   }
    …
…
```

This system is simple and easily tested and controlled for best results.

2 - Results and Reflection

Using a live tweaking tool (dat.GUI, 2012) and a lot of time to balance it, the AI system produced some very nice results; with hundreds of predators flowing around the environmental obstacles, separating and joining their neighbours in the swarm in a very natural-looking way; all at interactive rates – once switched to the basic Three.js WebGL renderer, the number of entities on-screen at a time at stable and interactive rates increased from about 400 to about 600.

Due to prioritising the ray-tracing part of the project after achieving the basic physics and swarming behaviour, the AI behaviour trees and states were not implemented, since they would not be used until after that had been achieved – in any case, I felt the ray-tracing component was the more challenging and interesting aspect.

Overall, the results were very satisfying.

SECTION IV – RAY-TRACING

1 - Research and Implementation

The process of implementing ray-tracing was definitely the most challenging and interesting aspect of the project.
Although familiar with the theory of ray-tracing (Wikipedia, 2012; Rademacher, 2012; Bikker, 2004), at the time of starting this project, I had little knowledge of the deeper implementation issues surrounding it – some of these issues were (as one person put it) "unknowable unknowns", in that they could not reasonably have been anticipated before beginning the work itself.
In order to determine the best approach to take, I researched the field extensively (Kay et al, 1986; Roger et al, 2007; Wald et al, 2001), comparing different methods and trying to assess their suitability with the restrictions of the platform in question and the requirements of the project taken into account.
I also closely examined existing implementations of ray-tracing, particularly the work of Evan Wallace (WebGL Path Tracing, 2010; WebGL Water, 2012), and any other information I could find.
My primary source – and apparently the most suitable approach for the project, with adaptations – became the work of Purcell et al (2002).
The practical side of my implementation was also guided in part by Wallace's – aside from some differences, such as that he had "baked" the geometry information into the shaders, which would not suit this project, as it would require a recompilation every time something moved.

Further to this, I had no prior experience with GLSL or WebGL (and very little with a 3D graphics library of any kind), and so had to learn how to use both of them as well.

I chose to use Three.js (2012) to aid in the standard WebGL setup. It's a great framework, but in a turbulent state and very much still in development; and with next to no documentation. This necessitated reading through the source code and the work of others using it to figure out what was going on, and I am confident in saying that I am now extremely familiar with its inner workings.
In combination with the working draft specification (Khronos Group, 2012), using and reading through Three.js so closely was my path to learning how to use WebGL, and also many more widely applicable tools and patterns.

In the case of GLSL (actually GLSL ES), I relied for some time solely on the online testing sandboxes I was using (GLSL Sandbox, 2012; Shader Toy, 2012), a few very basic tutorials (Lewis (2), 2011), and existing implementations to learn the capabilities of the language – reliable documentation or references were hard to come by, and it was a while into development by the time I came across one that was useful (Khronos Group, 2006).
I gradually got used to the particulars of the pipeline – vertex and fragment shaders, uniforms, attributes and varying – and some of the concerns of hardware programming and parallelism – avoiding branching statements, and dividing the workload between the CPU and GPU well by

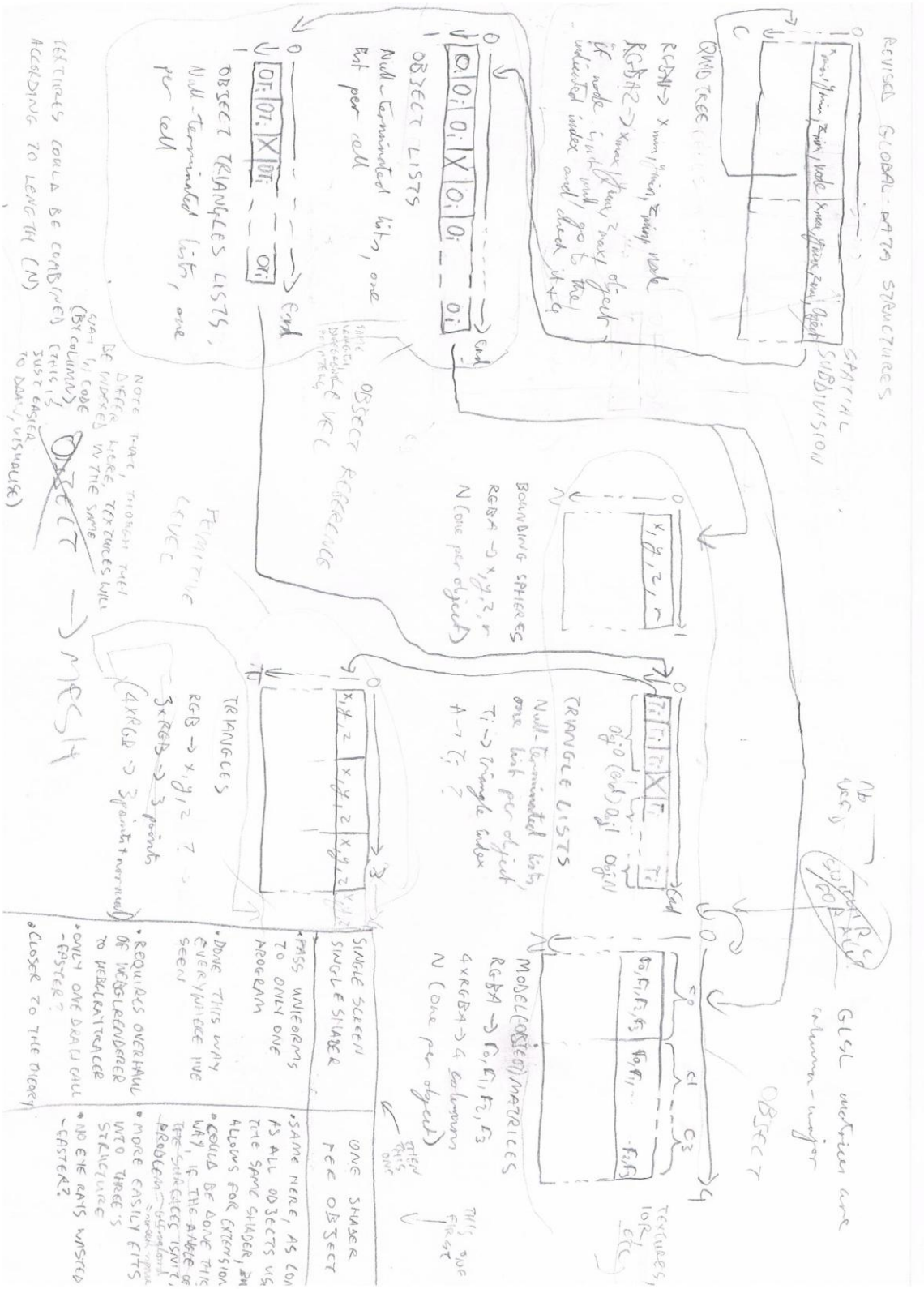
avoiding certain operations being performed per-pixel on the GPU. But, because it has a C-like syntax, the tendency was still there to attempt things with it as I would with a language that has more advanced logical capabilities.
Many of the language's limitations come from its aggressive optimisation of programs – such as allowing only a fixed-length loop, so it could be "unrolled" at compile-time for efficiency – but this sometimes extended further than I expected – for example, it doesn't allow variable-length arrays or array sizes to be non-constant values, but one particularly troublesome issue was that it doesn't even allow arrays to be indexed with values that aren't constant, loop-control-variables, or combinations thereof.

Finally, once I had a better idea of how to write GLSL, I came up against the issues surrounding the compiler and debugging GLSL programs.
GLSL is notoriously touchy about the code you write, and I discovered several instances (both anecdotally and in my own experience) where it failed to compile, apparently for no good reason. One intermittent issue occurred when trying to use a normal uniform float value. I could compare other values to it, and perform operations on it, but I could not use the uniform itself on its own (return it from a function, or assign it to a variable) – I had to do this by multiplying it by 10 and then dividing by 10, until the issue stopped arising (possibly something to do with the compiler optimising a value that is always directly referred to.
It is also very difficult to debug a GLSL program, as it doesn't output any useful runtime error information, and the only way of outputting information yourself is to draw a colour – this resulted in many long sessions of commenting out code and printing out colours based on an expected condition until something happened, inferring the cause of the problem, and then re-introducing code piece by piece.

1.1 - Version 1

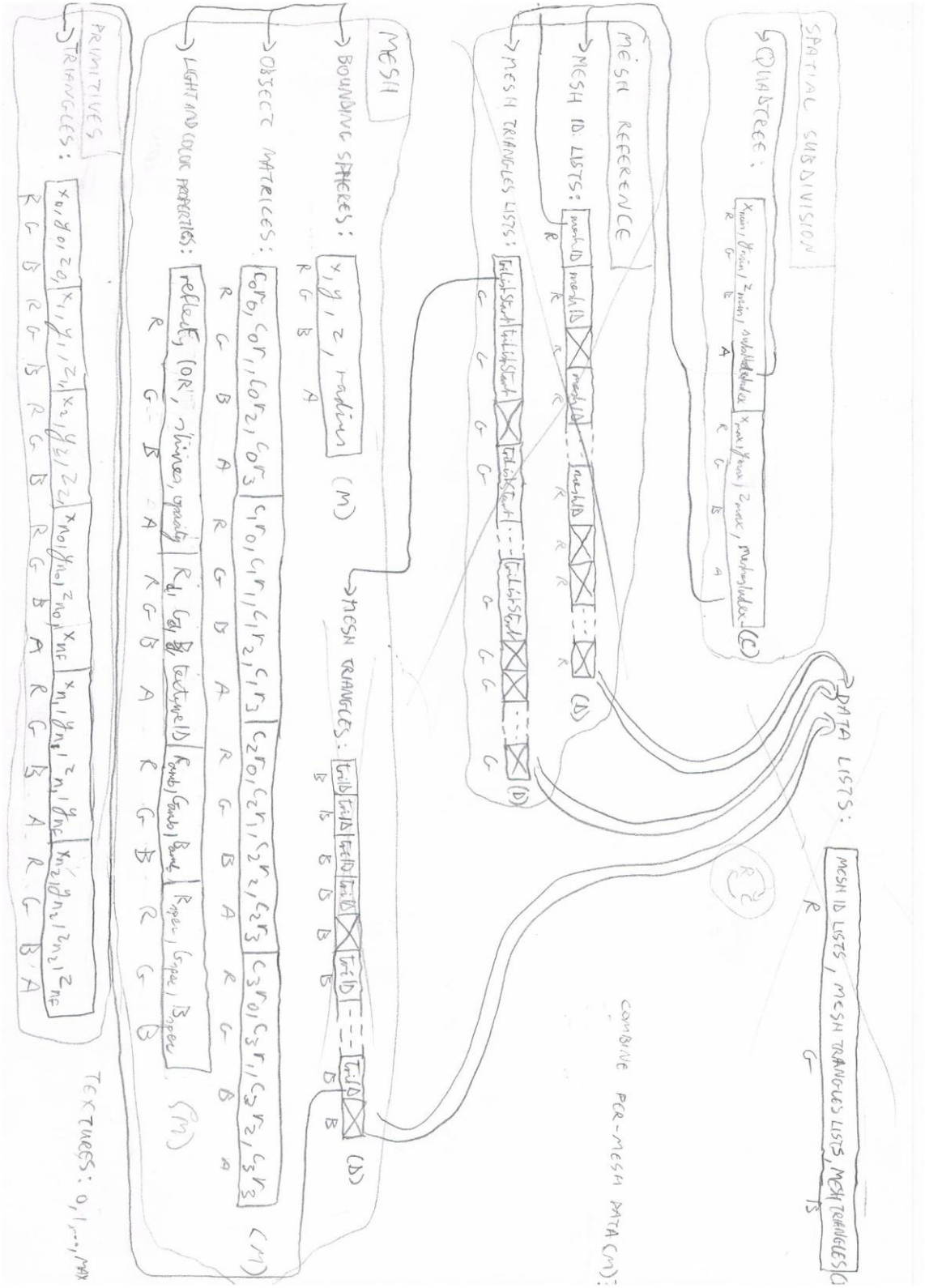
Because of all these issues, the learning process was a long and often difficult one. Once I had overcome the basic problems, I began work on the first version of the GPU ray-tracing program. Since I had examined Wallace's implementations very closely, knew what was happening, and was concerned about time, I decided to setup the bulk of the basic ray-tracer program, without spending time repeating what I had seen in others' working examples.
So I delved straight into it, testing what I could in the GLSL Sandbox as I went.

It was adapted from the system used by Purcell et al, expanding upon their method of passing global scene data to the GPU in textures (which have a far greater capacity and flexibility than standard GLSL arrays, and can have their sizes and indexes controlled via variables and uniforms, without recompiling).
The data-passing structure went through several iterations, beginning with nine data textures on paper, and finally arriving at three.



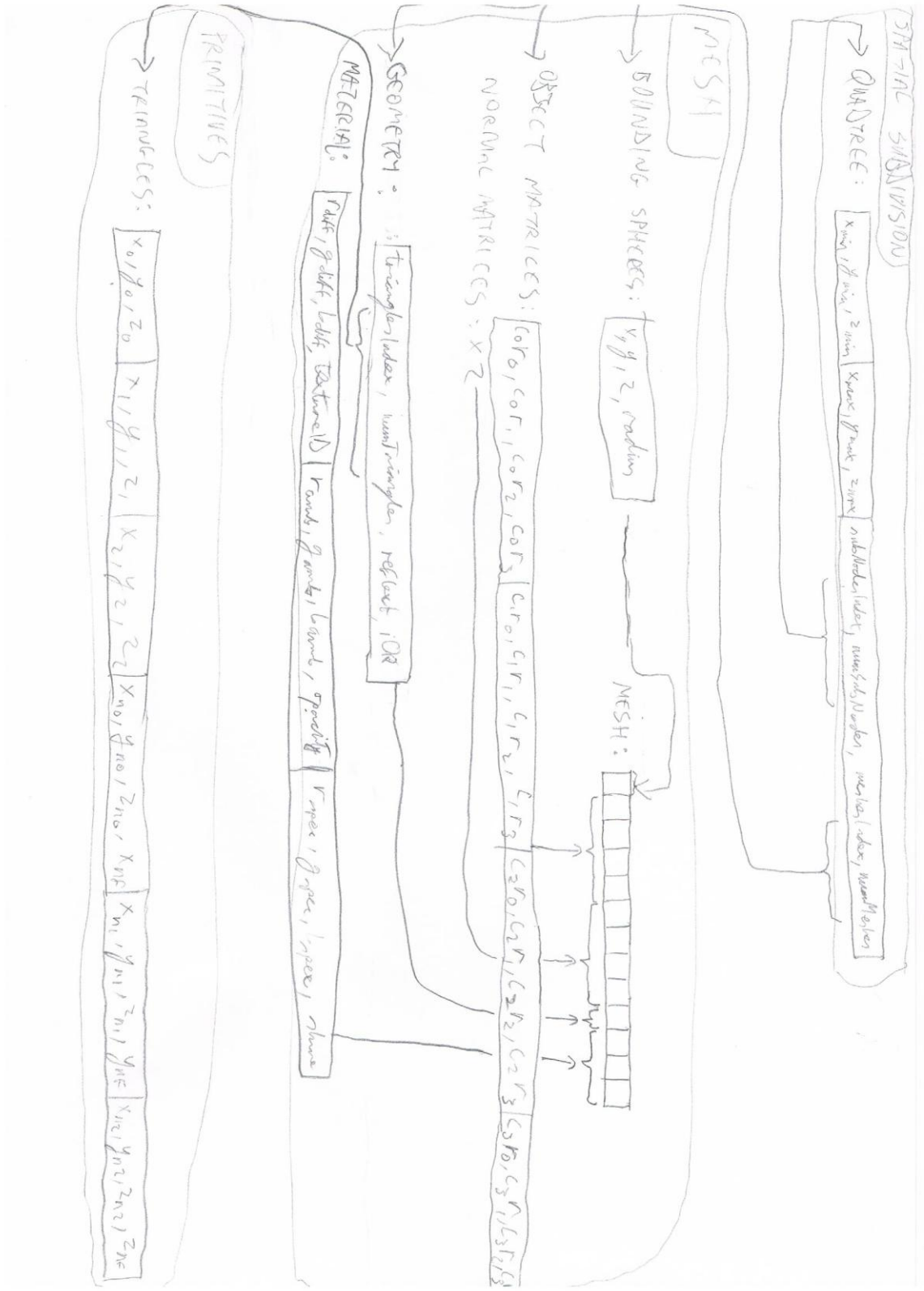
Data-passing version 1

9 textures (normal matrices not shown), with explicit pointers between tiers



Data-passing version 2

6 textures (lists packed into one one-dimensional RGBA texture), explicit pointers between tiers



Data-passing version 3

3 textures (each tier packed into one texture), implicit pointers between tiers (lists replaced by starting index and count, data sent from CPU in correct order, all unique one-to-many relationships)

Once the final data-passing strategy had been developed, the source code was written to construct the data textures, pass the data down, and read it back on the GPU side by querying the textures. The shader source code was written as strings in chunks, which were then concatenated and passed to Three.js to be built by WebGL:

- Structs
- Utility functions
- Data-reading functions
- Ray-object intersection tests (Rademacher, 2012; Blackpawn, 2012; Bogolepov, 2012)
- Ray-scene intersection test with faux-recursive quad-tree-traversal
- Whitted-style ray-tracer with reflection, refraction and shadow rays; phong colour calculation; light attenuation with maximum cutoffs (Madams, 2011) and spotlights (Lighthouse3D, 2012)

The code may be found at *Appendix A* (it's far too long to put here).

1.2 - Version 2

After sufficiently debugging version 1, it became apparent that the code was far too ambitious for GLSL ES:

- It required loops nested up to three deep, and since loops are unrolled, the system used to get around this meant each loop was very large
  - Set up a constant "budget" for each collection in the current build of the shader – this is the maximum number of items the collection can hold without requiring the shader to be recompiled
  - All loops iterating over the collection must loop over the entire collection – from zero to the budget minus 1 – because of the unrollable loop restriction of GLSL ES
  - Declare a variable counter, which iterates along with the loop counter, and break out of the loop if that counter exceeds your desired variable number of iterations
- GLSL ES doesn't support recursion of any kind, but it was required in this case, and so was faked by iterating over a flat array of items which was written to as the loop ran, and terminated by another flag – this resulted in a similarly large number of loops
- Other instructions were similarly overambitious – too much nesting, branching, etc.

Simply put, I was trying to make GLSL do things it didn't want to, and it was refusing to work. So, after quite a while chopping out complexity piece-by-piece, the ray-tracer arrived at version 2.

In this instance, the functionality had been simplified greatly, but finally worked:

- The nested loops were abandoned, and all intersections were tested against a flat array of global data

- The data itself had been reduced to bounding spheres, so everything was now spherical, apart from the backdrop, which was a special case plane
- The Whitted-style recursive ray-tracer had been simplified to a linear shadow-caster
    - One path to follow, as opposed to a tree of possible rays
    - One hit with the scene per ray
    - A single eye ray and a single shadow ray cast and intersected with the scene per-pixel
- Other possible but not verified problems may have been solved
    - The number of uniforms passed to the shader had been greatly reduced – GLSL has a maximum capacity for uniforms
    - The overall length of the program had been greatly reduced – GLSL can only manage a limited number of instructions, including those in unrolled loops

The code is available at *Appendix B*.

2 - Results and Reflection

The target of real-time ray-tracing was achieved, albeit not quite in the sense I had hoped for. It is greatly simplified from the ideal system, only supports spheres, and is really a shadow-caster. It also behaves slightly peculiarly from a theory point of view, and should be adapted to use uniforms passed for each mesh when processing the eye ray, as opposed to intersecting the eye ray with the entire scene – this means that some parts are drawn twice, and some geometry's rendering projected onto geometry behind it.
The compiler cannot handle more than a few meshes, which indicates that even in its current form, it's pressed up against the ceiling of what's possible on the current platform. Current compilation time on my machine is about 2 minutes.
It seems to have gone as far as it can.

However, it does work well for what it is.
The data-passing pipeline (reduced) works well and allows changes to be made on the fly, something I had not seen done in WebGL before.
Performance is smooth, and the results are (by definition) pixel-perfect, and quite easy on the eye. The visual theme of the game has – for the most part – been realised.

Reading back light hit information from the rendered frame was never addressed, but is possible – it would possibly involve encoding the hit information in an unused channel of the frame, reading the frame back on the CPU, moving to each entity's offset, and interpreting the data.

So, I am quite pleased to have something to show for the work that went into it – and glad to have learned as much as I did in the process.

SECTION V – REVIEW

As it stands, the project has addressed several of the more interesting technical challenges, and two working technical demonstrations have been produced – these show the spirit and potential of the ideal *Lumens* that could be.
I'm reasonably happy with the result, even if it isn't the perfect one I had secretly hoped for.

I would like to have a go at some of the areas I missed out on, such as basic game-play and environment generation using user-rated genetic algorithms.

In order to advance to the next phase, it is obvious that *Lumens* needs to be migrated to a more capable platform. WebGL is a young technology, and while it's amazing that there exists 3D graphics in web browsers at all, it isn't yet up to reaching this project's ideal targets.
The ceiling has well and truly been hit, in some sense – and I find a certain satisfaction in having pushed it to its limits.

The wish list for the next stage:

- An expanded instruction set and other capabilities on the GPU; or otherwise, a CPU capable of performing real-time ray-tracing at decent resolutions, with complex scenes, given a well-planned and written system (Bikker, 2008)
- A cross-platform language (probably C++) to keep the dream of a multi-platform release alive
- A more stable graphics library

Given these improvements and more development time, I would still hope to achieve most or all of the project's goals.

REFERENCES

Reynolds, Craig, 2001 (online), "*Boids: Background and Update*", red3d.com, accessed 10.10.2011, URL: http://www.red3d.com/cwr/boids/

Buckland, Mat, 2005, "Programming Game AI by Example", "Chapter 3: How to Create Autonomously Moving Game Agents: Group Behaviours"

Baldwin, Mark, 2005 (online), Baldwin Consulting, baldwinconsulting.org, accessed 15.10.2011, URL: www.baldwinconsulting.org

jQuery, 2012 (online), www.jquery.com, accessed 28.11.2011, URL: http://jquery.com/

Modernizr, 2012 (online), www.modernizr.com, accessed 28.11.2011, URL: http://www.modernizr.com/

HTML5 Boilerplate, 2012 (online), www.html5boilerplate.com, accessed 28.11.2011, URL: http://html5boilerplate.com/

Three.js, 2012 (online), github.com, accessed 3.2.2012, URL: https://github.com/mrdoob/three.js

Chambers, Mike, 2011 (online), *"JavaScript QuadTree Implementation"*, mikechambers.com, accessed 15.12.2011, URL: http://www.mikechambers.com/blog/2011/03/21/javascript-quadtree-implementation/

Mozilla Developer Network, 2012 (online), developer.mozilla.org, accessed 16.4.2012, URL: https://developer.mozilla.org/en-US/

Cabello, Ricardo, 2012 (online), mrdoob.com, accessed 16.4.2012, URL: http://mrdoob.com/

Irish, Paul, 2012 (online), paulirish.com, accessed 16.4.2012, URL: http://paulirish.com/

Hattab, Hakim El, 2012 (online), hakim.se, accessed 16.4.2012, URL: http://hakim.se/experiments

Lewis, Paul (1), 2012 (online), aerotwist.com, accessed 16.4.2012, URL: http://aerotwist.com/

Resig, John, 2012 (online), ejohn.org, accessed 16.4.2012, URL: http://ejohn.org/

Millington, Ian, 2007, *"Game Physics Engine Development"*, chapters 3-7, published by Morgan Kaufmann Publishers

Matyka, Maciej, 2004 (online), panoramix.ift.uni.wroc.pl/~maq/ , *"How to Implement a Pressure Soft Body Model"*, accessed 12.2.2012, URL: http://panoramix.ift.uni.wroc.pl/~maq/soft2d/howtosoftbody.pdf

Brundage, Harry, 2011 (online), harry.me, *"Neat Algorithms - Flocking"*, accessed 8.11.2011, URL: http://harry.me/2011/02/17/neat-algorithms---flocking

dat.GUI, 2012 (online), chromeexperiments.com, *"dat.GUI"*, accessed 8.3.2012, URL: http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage

Owen, G. Scott, 1999 (online), "*Ray Tracing*", siggraph.org, accessed 16.10.2011, URL: http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm

Wikipedia, 2012 (online), "*Ray Tracing (graphics)*", en.wikipedia.org, accessed 16.10.2011, URL: http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Rademacher, Paul, 2012 (online), *"Ray Tracing: Graphics for the Masses"*, cs.unc.edu, accessed 20.2.2012, URL: http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

Bikker, Jacco, 2004 (online), *"Raytracing Topics & Techniques"* (series), flipcode.com, accessed 15.3.2012, URL: http://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_1_Introduction.shtml

Kay, Timothy L.; Kajiya, James T.; 1986 (online), *"Ray Tracing Complex Scenes"*, cs.virginia.edu, accessed 8.3.2012, URL: http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Ray%20Tracing%20Complex%20Scenes.pdf

Roger, D.; Assarsson, U.; Holzschuh, N.; 2007 (online), *"Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU"*, maverick.inria.fr, accessed 4.4.2012, URL: http://maverick.inria.fr/Publications/2007/RAH07/RAH07Printed.pdf

Wald, Ingo; Slusallek, Philipp; 2001 (online), *"State of the Art in Interactive Ray Tracing"*, flipcode.net, accessed 20.3.2012, URL: http://www.flipcode.net/archives/State-of-the-Art%20in%20interactive%20ray%20tracing.pdf

WebGL Path Tracing, 2010 (online), developed by Evan Wallace, madebyevan.com, accessed 2.10.2012, URL: http://madebyevan.com/webgl-path-tracing/

WebGL Water, 2012 (online), developed by Evan Wallace, madebyevan.com, accessed 2.10.2012, URL: http://madebyevan.com/webgl-water/

Purcell, Timothy J.; Buck, Ian; Mark, William R.; Hanrahan, Pat; 2002 (online), *"Ray Tracing on Programmable Graphics Hardware"*, graphics.stanford.edu, accessed 16.4.2011, URL: http://graphics.stanford.edu/papers/rtongfx/rtongfx.pdf

Khronos Group, 2012 (online), *"WebGL Specification – Khronos Working Draft"*, khronos.org, accessed 2.3.2012, URL: http://www.khronos.org/registry/webgl/specs/latest/

GLSL Sandbox, 2012 (online), developed by Ricardo Cabello, mrdoob.com, accessed 14.2.2012, URL: http://mrdoob.com/projects/glsl_sandbox/

Shader Toy, 2012 (online), developed by Inigo Quilez, iquilezles.org, accessed 14.2.2012, URL: http://www.iquilezles.org/apps/shadertoy/

Lewis, Paul (2), 2011 (online), *"Shaders"* (series), aerotwist.com, accessed 14.2.2012, URL: http://aerotwist.com/tutorials/an-introduction-to-shaders-part-1/

Khronos Group, 2006 (online), *"The OpenGL ES Shading Language"*, khronos.org, accessed 10.4.2012, URL: http://www.khronos.org/files/opengles_shading_language.pdf

Blackpawn, 2012 (online), *"Point in triangle Test"*, blackpawn.com, accessed 7.4.2012, URL: http://www.blackpawn.com/texts/pointinpoly/default.html

Bogolepov, Dennis, 2012 (online), *"Implicit Surfaces Demo"*, code.google.com/p/rtrt-on-gpu, accessed 6.4.2012, URL: http://code.google.com/p/rtrt-on-gpu/source/browse/trunk/Source/GLSL+Tutorial/Implicit+Surfaces/Fragment.glsl?r=305

Madams, Tom, 2011 (online), *"Improved Light Attenuation"*, imdoingitwrong.wordpress.com, accessed 12.4.2012, URL: http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-attenuation/

Lightouse3D, 2012 (online), *"GLSL 1.2 Tutorial: Spot Light Per Pixel"*, lighthouse3d.com, accessed 10.4.2012, URL: http://www.lighthouse3d.com/tutorials/glsl-tutorial/spot-light-per-pixel/

Bikker, Jacco, 2008 (online), *"Architecture of a Real-Time Ray-Tracer"*, software.intel.com, accessed 11.4.2012, URL: http://software.intel.com/en-us/articles/architecture-of-a-real-time-ray-tracer/

APPENDIX

A

```
function RayTracer(options) {
        if(!options) { options = {}; }

        // Note: float texture values go from [-inf, +inf], they are
NOT normalised to [0, 1] -
http://stackoverflow.com/questions/5709023/what-exactly-is-a-
floating-point-texture
        this.uniforms = THREE.UniformsUtils.merge([
           THREE.UniformsLib["fog"],
           {
              ambientLightColor: { type: 'fv', value: [] },

              directionalLightColor: { type: 'fv', value: [] },
              directionalLightDirection: { type: 'fv', value: [] },

              pointLightColor: { type: 'fv', value: [] },
              pointLightPosition: { type: 'fv', value: [] },
              pointLightDistance: { type: 'fv1', value: [] },

              spotLightDirection: { type: 'fv', value: [] },
              spotLightCosAngle: { type: 'fv1', value: [] },
              spotLightAngle: { type: 'fv1', value: [] },
              spotLightFalloff: { type: 'fv1', value: [] },

              /* N*(2*RGB+1*RGBA) - [xmin, ymin, zmin][xmax, ymax,
zmax]
                 [subNodesIndex, numSubNodes, meshesIndex, numMeshes]
*/
              quadTree: { type: 't', value: 0,
                 texture: new THREE.DataTexture(new Float32Array(0),
                    0, 0, THREE.RGBAFormat, THREE.FloatType,
                    undefined, undefined, undefined,
                    THREE.NearestFilter, THREE.NearestFilter) },
              /*quadTree: { type: 't', value: 0,
                 texture: new THREE.DataTexture([], 0, 0,
                    THREE.RGBAFormat, THREE.IntType,
                    undefined, undefined, undefined,
                    THREE.NearestFilter, THREE.NearestFilter) },*/

              /* M*(2*RGB+15*RGBA - [boundRadius, trianglesIndex,
numTriangles]

                    [c0r0, c0r1, c0r2, c0r3][c1r0, c1r1, c1r2,
c1r3][c2r0, c2r1, c2r2, c2r3][c3r0, c3r1, c3r2, c3r3]
                    [c0r0, c0r1, c0r2, c0r3][c1r0, c1r1, c1r2,
c1r3][c2r0, c2r1, c2r2, c2r3][c3r0, c3r1, c3r2, c3r3]
```

```
                    [c0r0, c0r1, c0r2, c0r3][c1r0, c1r1, c1r2,
c1r3][c2r0, c2r1, c2r2, c2r3][c3r0, c3r1, c3r2, c3r3]

                    [reflect, iOR, density, opacity]
                    [ramb, gamb, bamb]
                    [rdiff, gdiff, bdiff, textureID]
                    [rspec, gspec, bspec, shine] */
               meshes: { type: 't', value: 1,
                  texture: new THREE.DataTexture(new Float32Array(0),
                     0, 0, THREE.RGBAFormat, THREE.FloatType,
                     undefined, undefined, undefined,
                     THREE.NearestFilter, THREE.NearestFilter) },
               /*meshes: { type: 't', value: 1,
                  texture: new THREE.DataTexture([], 0, 0,
                     THREE.RGBAFormat, THREE.IntType,
                     undefined, undefined, undefined,
                     THREE.NearestFilter, THREE.NearestFilter) },*/

               /* T*(3*RGB+3*RGBA) - [xp0, yp0, zp0][xp1, yp1,
zp1][xp2, yp2, zp2]
                  [xnp0, ynp0, znp0, xnf][xnp1, ynp1, znp1, ynf][xnp2,
ynp2, znp2, znf] */
               triangles: { type: 't', value: 2,
                  texture: new THREE.DataTexture(new Float32Array(0),
                     0, 0, THREE.RGBAFormat, THREE.FloatType,
                     undefined, undefined, undefined,
                     THREE.NearestFilter, THREE.NearestFilter) },
               /*triangles: { type: 't', value: 2,
                  texture: new THREE.DataTexture([], 0, 0,
                     THREE.RGBAFormat, THREE.IntType,
                     undefined, undefined, undefined,
                     THREE.NearestFilter, THREE.NearestFilter) },*/

               numNodes: { type: 'f', value: 0 },
               numMeshes: { type: 'f', value: 0 },
               numTriangles: { type: 'f', value: 0 },

               infinity: { type: 'f', value: (options.infinity ||
10000) },

               maxHits: { type: 'i', value: 0 },
               maxRays: { type: 'i', value: 0 }
            }]);

         this.material = new THREE.ShaderMaterial({
            fragmentShader: '', vertexShader: this.vertex,
            uniforms: this.uniforms, lights: true, fog: true
         });

         this.scene = options.scene;
         this.entities = options.entities;
```

```
        var b = options.budgets;
        this.budgets = {
           autoSetup: false,
           growth: 1.5,

           lights: this.scene.lights.length,
           meshes: (($.isNumeric(b.meshes))?
              b.meshes : this.entities.source.length),
           triangles: (($.isNumeric(b.triangles))?
              b.triangles : 100*b.meshes),
           hit: 0, ray: 0, quadTree: 0
        };

        this.setQuadTreeBudget(($.isNumeric(b.quadTreeDepth))?
              b.quadTreeDepth : this.entities.root.maxDepth)
           .setHitBudget(b.hit || 3)
           .setHitMax(options.maxHits || 3)
           .update().setup();

        this.budgets.autoSetup = !!b.autoSetup;
     }
     $.extend(RayTracer.prototype, {
        setup: function() {
           var b = this.budgets;

           this.material.fragmentShader =
              '#define HIT_BUDGET '+b.hit+'\n\
              #define HIT_BUDGET_F '+b.hit+'.0\n\
              #define RAY_BUDGET '+b.ray+'\n\
              #define RAY_BUDGET_F '+b.ray+'.0\n\
              #define QUAD_TREE_BUDGET '+b.quadTree+'\n\
              #define QUAD_TREE_BUDGET_F '+b.quadTree+'.0\n\
              #define MESHES_BUDGET '+b.meshes+'\n\
              #define MESHES_BUDGET_F '+b.meshes+'.0\n\
              #define TRIANGLES_BUDGET '+b.triangles+'\n\
              #define TRIANGLES_BUDGET_F '+b.triangles+'.0\n\n'+

              this.fragmentHeader+'\n'+this.structs+'\n'+this.util+
              '\n'+this.readData+'\n'+this.intersections+'\n'+
              this.intersectMeshes+'\n'+this.intersectScene+'\n'+
              this.shadowRayTracer+'\n'+this.fragmentMain;

           // Dirty material(?)
           this.refresh();

           return this;
        },
        update: function() { return
this.updateDataTextures().updateLights(); },
        updateDataTextures: function() {
```

```
            var u = this.uniforms,

               quadTreeTexture = u.quadTree.texture,
               meshesTexture = u.meshes.texture,
               trianglesTexture = u.triangles.texture,

               nodesData = [], meshesData = [], trianglesData = [],

               node = this.entities.root, nodes = [node],
               numMeshes = 0, numTriangles = 0,

               inverseObjectMatrix = new THREE.Matrix4(),
               normalMatrix = new THREE.Matrix4();

            /* TODO: fill all possible space for the QuadTree texture,
               to make recursion using the flattened array and
               constant expressions giving correct offsets for
               the next tier */
             // Quadtree tier
             for(var n = 0; n < nodes.length; node = nodes[++n]) {
               var kids = node.kids.concat(node.edgeKids),
                  minZ = Infinity, maxZ = -Infinity;

               // Mesh tier
               for(var k = 0; k < kids.length; ++k) {
                  var shape = kids[k].item.shape,
                     mesh = shape.three,
                     geometry = mesh.geometry,
                     vertices = geometry.vertices,
                     faces = geometry.faces;

                  // Primitive tier
                  for(var f = 0; f < faces.length; ++f) {
                     /* Assumes all faces are faces3 (ensure
                        exporting only triangles, or see
                        THREE.GeometryUtils.triangulateQuads)
                        and that all faces share the same material */
                     var face = faces[f],
                        a = vertices[face.a].position,
                        b = vertices[face.b].position,
                        c = vertices[face.c].position,

                        points = [a, b, c],

                        vertexNormals = face.vertexNormals,
                        an = vertexNormals[0],
                        bn = vertexNormals[1],
                        cn = vertexNormals[2],
                        fn = face.normal;

                     for(var p = 0; p < points.length; ++p) {
```

```
                        var point = points[p];

                        if(point.z < minZ) { minZ = point.z; }
                        if(point.z > maxZ) { maxZ = point.z; }
                     }

                     // TODO: vertex colors
                     trianglesData.push(a.x, a.y, a.z, 0,
                        b.x, b.y, b.z, 0,
                        c.x, c.y, c.z, 0,
                        an.x, an.y, an.z, fn.x,
                        bn.x, bn.y, bn.z, fn.y,
                        cn.x, cn.y, cn.z, fn.z);
                  }

                  var bRad = shape.boundRad.rad,

                     flatInverseObjectMatrix = [],
                     flatObjectMatrix = [],
                     flatNormalMatrix = [], // Already provided by
THREE, but as a 3x3 matrix, which won't fit so nicely here

                     mat = shape.material,
                     diff = mat.diffuse,
                     amb = mat.ambient,
                     spec = mat.specular;

                  inverseObjectMatrix.getInverse(mesh.matrixWorld)
                     .flattenToArray(flatInverseObjectMatrix);

                  mesh.matrixWorld.flattenToArray(flatObjectMatrix);

                 normalMatrix.getInverse(mesh.matrixWorld).transpose()
                     .flattenToArray(flatNormalMatrix);

                  meshesData = meshesData.concat(
                     bRad, ((faces.length)? numTriangles : -1),
faces.length, 0,
                     flatInverseObjectMatrix,
                     flatObjectMatrix,
                     flatNormalMatrix,
                     mat.reflect, mat.iOR, mat.density, mat.opacity,
                     amb.r, amb.g, amb.b, 0,
                     diff.r, diff.g, diff.b, mat.textureID,
                     spec.r, spec.g, spec.b, mat.shine);

                  numTriangles += faces.length;
               }

               var min = node.boundRect.pos, max =
node.boundRect.size;
```

```
               nodesData.push(min.x, min.y, minZ, 0,
                  max.x, max.y, maxZ, 0,
                  ((node.nodes.length)? nodes.length : -1),
node.nodes.length,
                     ((kids.length)? numMeshes : -1), kids.length);

               numMeshes += kids.length;
               nodes = nodes.concat(node.nodes);
            }

            // Note: OpenGL texture indices start at the bottom-left,
and are row-major
            quadTreeTexture.image.height =
               u.numNodes.value = nodes.length;
            quadTreeTexture.image.width = ((nodes.length)? 3 : 0);
            quadTreeTexture.image.data = new Float32Array(nodesData);

            meshesTexture.image.height = u.numMeshes.value =
numMeshes;
            meshesTexture.image.width = ((numMeshes)? 17 : 0);
            meshesTexture.image.data = new Float32Array(meshesData);

            trianglesTexture.image.height = u.numTriangles.value =
numTriangles;
            trianglesTexture.image.width = ((numTriangles)? 6 : 0);
            trianglesTexture.image.data = new
Float32Array(trianglesData);

            quadTreeTexture.needsUpdate =
               meshesTexture.needsUpdate =
                  trianglesTexture.needsUpdate = true;

            var bud = this.budgets;

            if(bud.autoSetup) {
               var setup = false;

               if(nodes.length > bud.quadTree) {
                  this.setQuadTreeBudget(this.entities.root.depth);
               }
               if(numMeshes > bud.meshes) {
                  bud.meshes = Math.floor(numMeshes*bud.growth);
                  setup = true;
               }
               if(numTriangles > bud.triangles) {
                  bud.triangles = Math.floor(numTriangles*bud.growth);
                  setup = true;
               }

               if(setup) { this.setup(); }
```

```
            }

            return this;
         },
         updateLights: function() {
            /* Note: updating the number and types of lights can't be
done without
               rebuilding the shader program - set
material.needsUpdate to true
               (can't find this in the source though - delete
material.program instead?)
               https://github.com/mrdoob/three.js/wiki/Updates

               An alternative may be to set it up a budgeted number
of lights with
               dummy information (0 intensity)

               Then again, this could be out of date entirey... */
            var u = this.uniforms,
               d = u.spotLightDirection.value,
               ca = u.spotLightCosAngle.value,
               a = u.spotLightAngle.value,
               f = u.spotLightFalloff.value,

               lights = this.scene.lights, ll = lights.length;

            for(var l = 0, vecOffset = 0; l < ll; ++l) {
               var light = lights[l];

               if(light instanceof SpotLight) {
                  d[vecOffset++] = light.direction.x;
                  d[vecOffset++] = light.direction.y;
                  d[vecOffset++] = light.direction.z;

                  ca[l] = light.cosAngle;
                  a[l] = Math.degrees(light.angle);
                  f[l] = light.falloff;
               }
               else {
                 d[vecOffset++] = d[vecOffset++] = d[vecOffset++] = 0;
                  ca[l] = f[l] = -1;
               }
            }

            if(this.budgets.autoSetup && ll > this.budgets.lights) {
               this.budgets.lights = ll;
               this.setup();
            }

            return this;
         },
```

```
        setQuadTreeBudget: function(depth) {
           this.budgets.quadTree = (Math.pow(4, depth)-1)/3;

           return this;
        },
        setHitBudget: function(hits) {
           this.budgets.hit = hits;
           this.budgets.ray = this.numRays(hits);

           return this;
        },
        setHitMax: function(hits) {
           this.uniforms.maxHits.value = hits;
           this.uniforms.maxRays.value = this.numRays(hits);

           if(this.budgets.autoSetup && hits > this.budgets.hit) {
              this.budgets.lights = this.scene.lights.length;
              this.setHitBudget(hits).setup();
           }

           return this;
        },
        numRays: function(hits) {
           // TODO: reflect and refract shadow rays for caustics etc
           return 1+(2+this.scene.lights.length)*
                 (Math.pow(2, hits)-1);
        },
        refresh: function() {
           this.material.needsUpdate = true;
           //delete this.material.program;

           return this;
        },

        /* Supplied by THREE:
           uniform mat4 objectMatrix;
           uniform mat4 modelViewMatrix;
           uniform mat4 projectionMatrix;
           uniform mat4 viewMatrix;
           uniform mat3 normalMatrix;
           uniform vec3 cameraPosition;

           attribute vec3 position;
           attribute vec3 normal;
           attribute vec2 uv;
           attribute vec2 uv2;

           as well as color, morph targets, and skinning
```

https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGLRenderer.js#L5273

```
            Set material.shading to THREE.SmoothShading to get
            per-vertex normals:

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Renderer.js#L774 */
         vertex:
            // For passing interpolated position to fragment
            'varying vec4 pos;\n\
            //varying vec3 norm;\n\
            \n\
            void main() {\n\
               pos = (objectMatrix*vec4(position, 1.0));\n\
               //norm = (normalMatrix*normal);\n\
               gl_Position = projectionMatrix*modelViewMatrix*\n\
                     vec4(position, 1.0);\n\
            }',

         /* Supplied by THREE:
            uniform mat4 viewMatrix;
            uniform vec3 cameraPosition;

            as well as precision

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Renderer.js#L5362

            Set material.lights to true to use lights and include the
            corresponding variables:

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Shaders.js#L419,

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Shaders.js#L458 */
         fragmentHeader:
            THREE.ShaderChunk["fog_pars_fragment"]+'\n\n'+
            /* TODO: use uniforms for eye ray */

            // toExponential ensures a float representation
            '#define PAD 0.25\n\
            \n\
            #define NODE_STEP '+(1.0/3.0).toExponential()+'\n\
            #define NODE_PAD '+(0.25/3.0).toExponential()+'\n\
            \n\
            #define MESH_DATA_STEP '+(1.0/17.0).toExponential()+'\n\
            #define MESH_DATA_PAD '+(0.25/17.0).toExponential()+'\n\
            #define MESH_OFFSET 0.0\n\
            #define INVERSE_OBJECT_MATRIX_OFFSET
'+(1.0/17.0).toExponential()+'\n\
```

```
            #define OBJECT_MATRIX_OFFSET
'+(5.0/17.0).toExponential()+'\n\
            #define NORMAL_MATRIX_OFFSET
'+(9.0/17.0).toExponential()+'\n\
            #define MATERIAL_OFFSET
'+(14.0/17.0).toExponential()+'\n\
            \n\
            #define TRI_POINT_STEP '+(1.0/6.0).toExponential()+'\n\
            #define TRI_POINT_PAD '+(0.25/6.0).toExponential()+'\n\
            \n\
            #define MEDIUM_IOR 1.0\n\
            #define EPSILON 0.01\n\
            \n\
            uniform vec3 ambientLightColor;\n\
            \n\
            #if MAX_POINT_LIGHTS > 0\n\
               uniform vec3 pointLightColor[MAX_POINT_LIGHTS];\n\
               uniform vec3 pointLightPosition[MAX_POINT_LIGHTS];\n\
               uniform float pointLightDistance[MAX_POINT_LIGHTS];\n\
               \n\
               uniform vec3 spotLightDirection[MAX_POINT_LIGHTS];\n\
               uniform float spotLightCosAngle[MAX_POINT_LIGHTS];\n\
               uniform float spotLightAngle[MAX_POINT_LIGHTS];\n\
               uniform float spotLightFalloff[MAX_POINT_LIGHTS];\n\
            #endif\n\
            #if MAX_DIR_LIGHTS > 0\n\
               uniform vec3 directionalLightColor[MAX_DIR_LIGHTS];\n\
               uniform vec3
directionalLightDirection[MAX_DIR_LIGHTS];\n\
            #endif\n\
            \n\
            /* Global scene data stored in textures */\n\
            uniform sampler2D quadTree;\n\
            uniform sampler2D meshes;\n\
            uniform sampler2D triangles;\n\
            \n\
            uniform float numNodes;\n\
            uniform float numMeshes;\n\
            uniform float numTriangles;\n\
            \n\
            uniform float infinity;\n\
            \n\
            uniform int maxHits;\n\
            uniform int maxRays;\n\
            \n\
            varying vec4 pos;\n\
            //varying vec3 norm;',

         structs:
            'struct Ray {\n\
               vec3 origin;\n\
```

```
               vec3 dir;\n\
               \n\
               int hit; /* Records the number of times this ray has
bounced in the scene */\n\
               \n\
               vec3 light; /* The light being transmitted (color,
energy) */\n\
               float iOR; /* The index of refraction of the medium
the ray is currently travelling through */\n\
               float tLight; /* The distance to the target light -
set for shadow rays, -1.0 otherwise */\n\
            };\n\
            \n\
            struct AABox {\n\
               vec3 min;\n\
               vec3 max;\n\
            };\n\
            \n\
            struct Node {\n\
               AABox bounds;\n\
               float subNodesIndex;\n\
               float numSubNodes;\n\
               float meshesIndex;\n\
               float meshCount;\n\
            };\n\
            \n\
            struct Mesh {\n\
               float boundRad;\n\
               \n\
               float trianglesIndex;\n\
               float triangleCount;\n\
            };\n\
            \n\
            struct Triangle {\n\
               vec3 points[3];\n\
               vec3 normals[3];\n\
               vec3 faceNormal;\n\
            };\n\
            \n\
            struct Material {\n\
               float reflect;\n\
               float iOR;\n\
               float density;\n\
               float opacity;\n\
               \n\
               vec3 ambient;\n\
               \n\
               vec3 diffuse; /* If textureID is valid, multiply texel
by diffuse, if not, just use diffuse */\n\
               int textureID;\n\
               \n\
```

```
            vec3 specular;\n\
            float shine;\n\
         };',

      util:
         'float sum(vec3 v) { return v.x+v.y+v.z; }\n\
         \n\
         bool inRange(float t) { return (EPSILON <= t && t <
infinity); }\n\
         \n\
         Ray transform(Ray ray, mat4 matrix) {\n\
            return Ray((matrix*vec4(ray.origin, 1.0)).xyz,\n\
               normalize((matrix*vec4(ray.dir, 0.0)).xyz),  /* Does
this need to be normalised again? */\n\
               ray.hit, ray.light, ray.iOR, ray.tLight);\n\
         }\n\
         \n\
         Triangle transform(Triangle tri, mat4 objMat, mat4
normMat) {\n\
            Triangle t;\n\
            \n\
            for(int p = 0; p < 3; ++p) {\n\
               t.points[p] = (objMat*vec4(tri.points[p],
1.0)).xyz;\n\
            }\n\
            for(int n = 0; n < 3; ++n) {\n\
               t.normals[n] =
normalize((normMat*vec4(tri.normals[n], 0.0)).xyz);  /* Does this
need to be normalised again? */\n\
            }\n\
            t.faceNormal = normalize((normMat*vec4(tri.faceNormal,
0.0)).xyz);\n\
            \n\
            return t;\n\
         }',

      /* Functions for reading in data from the respective
textures
         and returning a useable object */
      readData:
         'Node getNode(float index) {\n\
            float i = NODE_PAD,\n\
               j = (index+PAD)/numNodes;\n\
            \n\
            AABox bounds = AABox(texture2D(quadTree, vec2(i,
j)).xyz,\n\
                  texture2D(quadTree, vec2(i += NODE_STEP,
j)).xyz);\n\
            \n\
            vec4 pointers = texture2D(quadTree,\n\
               vec2(i += NODE_STEP, j));\n\
```

```
            \n\
          return Node(bounds, pointers.x, pointers.y, pointers.z,
pointers.w);\n\
            /*return Node(AABox(vec3(1.0), vec3(1.0)), 0.0, 0.0,
0.0, 1.0);*/\n\
         }\n\
         \n\
         Mesh getMesh(float index) {\n\
            vec4 data = texture2D(meshes,\n\
                  vec2(MESH_OFFSET+MESH_DATA_PAD,\n\
                     (index+PAD)/numMeshes));\n\
            \n\
            return Mesh(data.x, data.y, data.z);\n\
            /*return Mesh(0.0, 0.0, 0.0);*/\n\
         }\n\
         \n\
         mat4 getMatrix(float offset, float index) {\n\
            float i = offset+MESH_DATA_PAD,\n\
               j = (index+PAD)/numMeshes;\n\
            \n\
            // Note: OpenGL matrices are column-major - weird, be
careful\n\
            return mat4(texture2D(meshes, vec2(i, j)),\n\
               texture2D(meshes, vec2(i += MESH_DATA_STEP, j)),\n\
               texture2D(meshes, vec2(i += MESH_DATA_STEP, j)),\n\
               texture2D(meshes, vec2(i += MESH_DATA_STEP, j)));\n\
         }\n\
         \n\
         /* Multiply the ray (in world space) by the inverse of
the\n\
            transformation matrix when testing for
intersections,\n\
            to reduce the number of multiplications */\n\
         mat4 getInverseObjectMatrix(float index) {\n\
            return getMatrix(INVERSE_OBJECT_MATRIX_OFFSET,
index);\n\
         }\n\
         \n\
         /* If there\'s a hit, move the point back into world
space\n\
            by multiplying it by the transformation matrix */\n\
         mat4 getObjectMatrix(float index) {\n\
            return getMatrix(OBJECT_MATRIX_OFFSET, index);\n\
         }\n\
         \n\
         /* Normals must be handled as well, once a hit is
found,\n\
            and must be multiplied by the normal matrix - the\n\
            transpose of the inverse of the transformation
matrix,\n\
            with w = 0 (to remove translation)\n\
```

```
  http://www.unknownroad.com/rtfm/graphics/rt_normals.html */\n\
          mat4 getNormalMatrix(float index) {\n\
             return getMatrix(NORMAL_MATRIX_OFFSET, index);\n\
          }\n\
          \n\
          Material getMaterial(float index) {\n\
             float i = MATERIAL_OFFSET+MESH_DATA_PAD,\n\
                j = (index+PAD)/numMeshes;\n\
             \n\
             vec4 data[4];\n\
             \n\
             data[0] = texture2D(meshes, vec2(i, j));\n\
             data[1] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
             data[2] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
             data[3] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
             \n\
             return Material(data[0].r, data[0].g, data[0].b,
data[0].a,\n\
                data[1].rgb,\n\
                data[2].rgb, int(data[2].a), data[3].rgb,
data[3].a);\n\
          }\n\
          \n\
          Triangle getTriangle(float index) {\n\
             float i = TRI_POINT_PAD,\n\
                j = (index+PAD)/numTriangles;\n\
             Triangle t;\n\
             \n\
             t.points[0] = texture2D(triangles, vec2(i, j)).xyz;\n\
             t.points[1] = texture2D(triangles,\n\
                   vec2(i += TRI_POINT_STEP, j)).xyz;\n\
             t.points[2] = texture2D(triangles,\n\
                   vec2(i += TRI_POINT_STEP, j)).xyz;\n\
             \n\
             /* Note that the normal is transformed by a
different\n\
                normal matrix - this is in eye space in THREE, so\n\
                how do we get it down in world space?\n\
                vec3 nWorld = mat3(objectMatrix[0].xyz,
objectMatrix[1].xyz, objectMatrix[2].xyz)*normal;\n\
                Does this hold up when scaling is involved
though?\n\
                http://www.lighthouse3d.com/tutorials/glsl-
tutorial/the-normal-matrix/ */\n\
             vec4 n[3];\n\
             \n\
             n[0] = texture2D(triangles,\n\
```

```
                  vec2(i += TRI_POINT_STEP, j));\n\
               n[1] = texture2D(triangles,\n\
                  vec2(i += TRI_POINT_STEP, j));\n\
               n[2] = texture2D(triangles,\n\
                  vec2(i += TRI_POINT_STEP, j));\n\
               \n\
               t.normals[0] = n[0].xyz;\n\
               t.normals[1] = n[1].xyz;\n\
               t.normals[2] = n[2].xyz;\n\
               \n\
               t.faceNormal = vec3(n[0].w, n[1].w, n[2].w); //
Passing the face normal this way saves space\n\
               \n\
               return t;\n\
            }',

         /* The returned float (t) denotes the distance along
            the ray at which the intersection occurs, where valid
values
            for t are in the range [epsilon, infinity], exclusively
*/
         intersections:
            '/* Kajiya et al\n\
               http://code.google.com/p/rtrt-on-
gpu/source/browse/trunk/Source/GLSL+Tutorial/Implicit+Surfaces/Fragm
ent.glsl?r=305#147\n\
               http://www.gamedev.net/topic/495636-raybox-collision-
intersection-point/ */\n\
            float intersection(Ray ray, AABox box) {\n\
               float t = -1.0;\n\
               \n\
               /* Cater for the special case where the ray
originates\n\
                  inside the box - this must be handled explicitly
*/\n\
               if(all(greaterThan(ray.origin, box.min)) &&\n\
                  all(lessThan(ray.origin, box.max))) { t =
EPSILON; }\n\
               else {\n\
                  vec3 tmin = (box.min-ray.origin)/ray.dir;\n\
                  vec3 tmax = (box.max-ray.origin)/ray.dir;\n\
                  \n\
                  vec3 rmax = max(tmax, tmin);\n\
                  vec3 rmin = min(tmax, tmin);\n\
                  \n\
                  float end = min(rmax.x, min(rmax.y, rmax.z));\n\
                  float start = max(max(rmin.x, 0.0), max(rmin.y,
rmin.z));\n\
                  \n\
                  if(end > start) { t = start; }\n\
                  //return final > start;\n\
```

```
            }\n\
            \n\
            return t;\n\
         }\n\
         \n\
         /* See "Intersecting a Sphere" at
http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html */\n\
         float intersection(Ray ray, float rad) {\n\
            vec3 oc = -ray.origin;\n\
            float v = dot(oc, ray.dir);\n\
            float dSq = rad*rad-(dot(oc, oc)-v*v);\n\
            \n\
            return ((dSq >= 0.0)? v-sqrt(dSq) : -1.0);\n\
         }\n\
         \n\
         /* Ray-plane test */\n\
         float intersection(Ray ray, vec3 point, vec3 normal) {\n\
            float d = dot(ray.dir, normal);\n\
            \n\
            // Less than or equal to zero if parrallel or behind
ray\n\
            return ((abs(d) >= EPSILON)?\n\
               dot(normal, (point-ray.origin))/d : -1.0);\n\
         }\n\
         \n\
         /* Barycentric coordinate test -
http://www.blackpawn.com/texts/pointinpoly/default.html */\n\
         float intersection(Ray ray, Triangle triangle, out vec3
hit) {\n\
            hit = ray.origin;   // Mark it invalid\n\
            float t = intersection(ray, triangle.points[0],\n\
               triangle.faceNormal);\n\
            \n\
            // Check if parallel to or behind ray\n\
            if(inRange(t)) {\n\
               vec3 p = ray.origin+ray.dir*t;\n\
               \n\
               // Compute vectors\n\
               vec3 v0 = triangle.points[2]-triangle.points[0];\n\
               vec3 v1 = triangle.points[1]-triangle.points[0];\n\
               vec3 v2 = p-triangle.points[0];\n\
               \n\
               // Compute dot products\n\
               float dot00 = dot(v0, v0);\n\
               float dot01 = dot(v0, v1);\n\
               float dot02 = dot(v0, v2);\n\
               float dot11 = dot(v1, v1);\n\
               float dot12 = dot(v1, v2);\n\
               \n\
               // Compute barycentric coordinates\n\
               float invDenom = 1.0/(dot00*dot11-dot01*dot01);\n\
```

```
                  float u = (dot11*dot02-dot01*dot12)*invDenom;\n\
                  float v = (dot00*dot12-dot01*dot02)*invDenom;\n\
                  \n\
                  // Check if point is in triangle\n\
                  if(u < 0.0 || v < 0.0 || u+v >= 1.0) {\n\
                     hit = p;\n\
                     t = -1.0;\n\
                  }\n\
               }\n\
               \n\
               return t;\n\
            }',

         intersectMeshes:
            '/* Finds the closest intersection along the ray with
the\n\
               referenced meshes, in object space - messy to leave
the\n\
               responsibility of transforming back to some other
function\n\
               that didn\'t apply the transformation in the first
place,\n\
               but it\'s quicker than doing that for each mesh */\n\
            float intersection(Ray ray, float meshesIndex, float
meshCount,\n\
               out float meshID, out Triangle triangle, out vec3 hit)
{\n\
               float closest = 10.0*infinity/10.0;\n\
               Triangle dummy;\n\
               triangle = dummy;\n\
               meshID = -1.0;\n\
               hit = vec3(0.0);\n\
               \n\
               float mID = meshesIndex;\n\
               \n\
               for(float m = 0.0; m < MESHES_BUDGET_F; ++m) {\n\
                  if(m < meshCount) {\n\
                     Mesh mesh = getMesh(mID);\n\
                     Ray tRay = transform(ray,\n\
                        getInverseObjectMatrix(mID));\n\
                     \n\
                     if(inRange(intersection(tRay, mesh.boundRad))) {
  /* Check mesh bounding radii */\n\
                        /* Check mesh triangles */\n\
                        float tn = mesh.trianglesIndex;\n\
                        \n\
                       for(float t = 0.0; t < TRIANGLES_BUDGET_F; ++t)
{\n\
                           if(t < mesh.triangleCount) {/* Check for
end of list */\n\
                              Triangle tri = getTriangle(tn);\n\
```

```
                              vec3 h;\n\
                              float tT = intersection(tRay, tri,
h);\n\
                              \n\
                              if(EPSILON <= tT && tT < closest) {\n\
                                 closest = tT;\n\
                                 meshID = mID;\n\
                                 triangle = tri;\n\
                                 hit = h;\n\
                              }\n\
                              \n\
                              ++tn;\n\
                           }\n\
                           else { break; }\n\
                        }\n\
                     }\n\
                     \n\
                     ++mID;\n\
                  }\n\
                  else { break; }  /* Check for end of list */\n\
               }\n\
               \n\
               return closest;\n\
            }',

         intersectScene:
            '/* Finds the closest intersection along the ray with
the\n\
               entire scene, in object space */\n\
            float intersection(Ray ray, out float meshID,\n\
               out Triangle triangle, out vec3 hit) {\n\
               float closest = 10.0*infinity/10.0;\n\
               Triangle dummy;\n\
               triangle = dummy;\n\
               meshID = -1.0;\n\
               hit = vec3(0.0);\n\
               \n\
               /* For the sake of optimisation, loops are
extremely\n\
                  primitive in GLSL - only constant values may be
used\n\
                  for the LCV, the test must be very simple, and
everything\n\
                  seems to need to be figured out at compile time,
not\n\
                  run time - hence, set up a budget of the maximum
possible\n\
                  number of loops, and break early if not all are
needed */\n\
               float nodes[QUAD_TREE_BUDGET];\n\
               nodes[0] = 0.0;\n\
```

```
               int nLast = 0;\n\
               \n\
               /* Traverse quadTree */\n\
               for(int n = 0; n < QUAD_TREE_BUDGET; ++n) {\n\
                  if(n <= nLast && float(n) < numNodes) {\n\
                     Node node = getNode(nodes[n]);\n\
                     bool intersects = inRange(intersection(ray,
node.bounds));\n\
                     \n\
                     if(intersects && node.numSubNodes > 0.0) {\n\
                        for(int nl = 0; nl < QUAD_TREE_BUDGET; ++nl)
{\n\
                           if(nl > nLast) { /* Super-ugly hack to get
the last element in the flat node list, since GLSL ES doesn\'t allow
non-constant expressions to be used to index an array (so no
"nodes[++nLast] = node.subNodesIndex", which is the nice way to do
it) */\n\
                              nodes[nl] = node.subNodesIndex;\n\
                              nodes[nl+1] = node.subNodesIndex+1.0;\n\
                              nodes[nl+2] = node.subNodesIndex+2.0;\n\
                              nodes[nl+3] = node.subNodesIndex+3.0;\n\
                              \n\
                              nodes[nl+4] = -1.0;\n\
                              nLast += 4;\n\
                              break;\n\
                           }\n\
                        }\n\
                     }\n\
                     \n\
                     if(intersects && node.meshCount > 0.0) {  /*
Check at every level to avoid rechecking borderKids by passing them
down to each child node */\n\
                        float mID;\n\
                        Triangle tri;\n\
                        vec3 h;\n\
                        float mT = intersection(ray,
node.meshesIndex,\n\
                              node.meshCount, mID, tri, h);\n\
                        \n\
                        if(mT < closest) {\n\
                           closest = mT;\n\
                           triangle = tri;\n\
                           meshID = mID;\n\
                           hit = h;\n\
                        }\n\
                     }\n\
                  }\n\
                  else { break; }\n\
               }\n\
               \n\
               return closest;\n\
```

```
            }',

         /* Given an initial ray, traces rays around the
            scene up to the maximum number allowed
            (maintaining recursive heirarchical order),
            and returns the accumulated color of the fragment */
         whittedRayTracer:
            'vec3 traceRays(Ray eyeRay) {\n\
               vec3 accColor;\n\
               \n\
               Ray rays[RAY_BUDGET];\n\
               \n\
               /* TODO: figure out if the eye ray needs to be
traced,\n\
                  or if nothing is between the fragment and the\n\
                  camera, since it is being drawn */\n\
               rays[0] = eyeRay;\n\
               \n\
               /* rLast - for reducing the number of loops\n\
                  When spawning rays, add them to rays after this\n\
                  and increase it by the number of added rays\n\
                  If r exceeds this, break (no more valid rays to
be\n\
                  traced) */\n\
               int rLast = 0;\n\
               \n\
               float meshID;\n\
               Triangle tri;\n\
               vec3 hit;\n\
               \n\
               for(int r = 0; r < RAY_BUDGET; ++r) {\n\
                  if(r <= rLast && r < maxRays) {\n\
                     Ray ray = rays[r];\n\
                     float t = intersection(ray, meshID, tri, hit);\n\
                     \n\
                     /* Calculate color from closest intersection
*/\n\
                     if(EPSILON <= ray.tLight && /* Shadow ray */\n\
                        (ray.tLight <= t || t < EPSILON)) {  /* Light
closer or no hit */\n\
                        /* TODO: caustics, reflected light\n\
                           Cast further reflected and refracted
shadow\n\
                           rays which accumulate color correctlty in
the\n\
                           case of an intersection with a
reflective\n\
                           or refractive mesh */\n\
                        /* Final color accumulation - direct light
*/\n\
                        accColor += ray.light;\n\
```

```
                    }\n\
                    else if(inRange(t) && ray.hit < maxHits) {\n\
                       /* Cast shadow, reflection, and\n\
                          refraction rays, if the energy is\n\
                          over the threshold */\n\
                       int h = ray.hit+1;\n\
                       \n\
                       mat4 objMat = getObjectMatrix(meshID);\n\
                       hit = (objMat*vec4(hit, 1.0)).xyz;\n\
                       Triangle triangle = transform(tri, objMat,
getNormalMatrix(meshID));\n\
                       \n\
                       /* TODO: get smooth normal -
http://www.codeproject.com/Articles/20144/Simple-Ray-Tracing-in-C-
Part-VI-Vertex-Normal-Inte */\n\
                       vec3 hitNormal = triangle.faceNormal;\n\
                       Material material = getMaterial(meshID);\n\
                       \n\
                       /* Set iOR according to whether leaving or\n\
                          entering material */\n\
                       /*if(dot(ray.dir, hitNormal) >= 0.0) {\n\
                          material.iOR = MEDIUM_IOR;\n\
                       }*/\n\
                       float leaving = ceil(dot(ray.dir,
hitNormal));\n\
                       material.iOR =
MEDIUM_IOR*leaving+material.iOR*(1.0-leaving);\n\
                       \n\
                       // Shadow\n\
                       vec3 shLight = material.opacity*ray.light*\n\
                            (material.ambient+material.diffuse+\n\
                            material.specular);\n\
                       \n\
                       if(sum(shLight) > 0.0) {\n\
                          /* Get the hit color */\n\
                          /* Branching is slow - only do it there\'s
a\n\
                             significant chunk of code inside */\n\
                          vec3 ambient =
ambientLightColor*material.ambient;\n\
                          \n\
                          for(int l = 0; l < MAX_POINT_LIGHTS; ++l)
{\n\
                             vec3 lightColor = pointLightColor[l];\n\
                             \n\
                             if(sum(lightColor) > 0.0) {\n\
                                vec3 toLight = pointLightPosition[l]-
hit;\n\
                                float dist = length(toLight);\n\
                                \n\
                                toLight /= dist;\n\
```

```
                                    \n\
                                    vec3 fromLight = -toLight;\n\
                                    \n\
                                    /* If falloff distance, determine
attenuation\n\

   http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                                    float attenuation = 1.0,\n\
                                       lightRange =
pointLightDistance[l];\n\
                                    \n\
                                    if(lightRange > 0.0) {\n\
                                       float distFactor = dist/\n\
                                             (1.0-pow(dist/lightRange,
2.0));\n\
                                       \n\
                                       attenuation =
1.0/pow(distFactor+1.0, 2.0);\n\
                                    }\n\
                                    \n\
                                    float lightCosAngle =
spotLightCosAngle[l];\n\
                                    \n\
                                    /* If spot light, determine if within
spot light cone */\n\
                                    if(attenuation > 0.0 &&\n\
                                       -1.0 < lightCosAngle &&
lightCosAngle < 1.0) { /* In the range [0-180], exclusive - should it
be [0, 1]->[90, 0]? (see attenuation with spot falloff) */\n\
                                       float cosAngle = dot(fromLight,\n\
                                          spotLightDirection[l]);\n\
                                       \n\
                                       attenuation *= ((lightCosAngle <=
cosAngle)?\n\
                                          /* Which will result in proper
attenuation?\n\
                                             http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\

   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
                                          pow(cosAngle,
spotLightFalloff[l]) : 0.0);\n\
                                          //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]) : 0.0);\n\
                                    }\n\
                                    \n\
                                    if(attenuation > 0.0) {   /* Don\'t
bother calculating color or casting if outside of light range */\n\
```

```
                                    /* Compute the phong shading at the
hit\n\
                                      Ambient is only accumulated once,
so do\n\
                                       that outside this loop */\n\
                                    vec3 rayColor;\n\
                                    \n\
                                    vec3 diffuse =
material.diffuse*lightColor*\n\
                                          max(dot(hitNormal, toLight),
0.0);\n\
                                    \n\
                                    /* TODO: texturing *//*\n\
                                    if(material.textureID >= 0) {\n\
                                       vec4 texel =
texture2D(textures[material.textureID],\n\
                                          getTextureCoord(meshID,
hit));\n\
                                       \n\
                                       diffuse *= texel.rgb*texel.a;\n\
                                    }*/\n\
                                    \n\
                                    rayColor += diffuse;\n\
                                    \n\
                                    if(sum(material.specular) > 0.0)
{\n\
                                       vec3 reflection =
reflect(toLight, hitNormal);\n\
                                       float highlight =
max(dot(reflection, ray.dir), 0.0);\n\
                                       vec3 specular =
material.specular*lightColor*\n\
                                             pow(highlight,
material.shine);\n\
                                       \n\
                                       rayColor += specular;\n\
                                    }\n\
                                    \n\
                                    rayColor *= material.opacity;\n\
                                    \n\
                                    if(sum(rayColor) > 0.0) { /* Don\'t
bother casting if it\'s just black anyway  */\n\
                                       /* TODO: take jittered samples
for soft shadows? */\n\
                                       Ray shadow = Ray(hit,\n\
                                          toLight, h,\n\

   shLight*rayColor*attenuation,\n\
                                          material.iOR, dist);\n\
                                       \n\
                                       float meshIDSh;\n\
```

```
                                    Triangle triangleSh;\n\
                                    vec3 hitSh;\n\
                                    \n\
                                    float tSh =
intersection(shadow,\n\
                                         meshIDSh, triangleSh,
hitSh);\n\
                                    \n\
                                    /* Calculate color from closest
intersection */\n\
                                    if(EPSILON <= shadow.tLight
&&\n\
                                       (ray.tLight <= tSh || tSh <
EPSILON)) {/* Light closer or no hit */\n\
                                       accColor += shadow.light;\n\
                                    }\n\
                                 }\n\
                              }\n\
                           }\n\
                        }\n\
                        \n\
                        accColor +=
shLight*ambient*material.opacity;\n\
                     }\n\
                     \n\
                     // Reflection\n\
                     vec3 rflLight =
material.reflect*material.opacity*ray.light;\n\
                     if(sum(rflLight) > 0.0) {\n\
                       for(int rl = 0; rl < QUAD_TREE_BUDGET; ++rl)
{\n\
                           if(rl > rLast) {\n\
                              rays[rl] = Ray(hit,\n\
                                 reflect(ray.dir, hitNormal), h,\n\
                                 rflLight, material.iOR, -1.0);\n\
                              \n\
                              rLast = rl;\n\
                              break;\n\
                           }\n\
                        }\n\
                     }\n\
                     \n\
                     // Refraction - TODO: Beer\'s Law -
http://www.flipcode.com/archives/Raytracing_Topics_Techniques-
Part_3_Refractions_and_Beers_Law.shtml\n\
                     vec3 rfrLight = (1.0-
material.opacity)*ray.light;\n\
                     if(sum(rfrLight) > 0.0) {\n\
                       for(int rl = 0; rl < QUAD_TREE_BUDGET; ++rl)
{\n\
                           if(rl > rLast) {\n\
```

```
                                rays[rl] = Ray(hit,\n\
                                   refract(ray.dir, hitNormal,\n\
                                      ray.iOR/material.iOR), h,\n\
                                   rfrLight, material.iOR, -1.0);\n\
                                \n\
                                rLast = rl;\n\
                                break;\n\
                             }\n\
                          }\n\
                       }\n\
                    }\n\
                 }\n\
                 else { break; }\n\
              }\n\
              \n\
              return accColor;\n\
           }',

        /* Linear ray tracer (doesn't branch into a tree),
           shadow rays */
        shadowRayTracer:
           'vec3 traceRays(Ray eyeRay) {\n\
              vec3 accColor;\n\
              Ray ray = eyeRay;\n\
              float meshID;\n\
              Triangle tri;\n\
              vec3 hit;\n\
              float t = intersection(ray, meshID, tri, hit); // TODO:
remove this intersection, pass each mesh\'s data in uniforms for
this instead \n\
              \n\
              if(inRange(t)) {\n\
                 /* Cast shadow ray, if the energy is over the
threshold */\n\
                 int h = ray.hit+1;\n\
                 \n\
                 mat4 objMat = getObjectMatrix(meshID);\n\
                 hit = (objMat*vec4(hit, 1.0)).xyz;\n\
                 Triangle triangle = transform(tri, objMat,
getNormalMatrix(meshID));\n\
                 \n\
                 /* TODO: get smooth normal */\n\
                 vec3 hitNormal = triangle.faceNormal;\n\
                 Material material = getMaterial(meshID);\n\
                 \n\
                 float leaving = ceil(dot(ray.dir, hitNormal));\n\
                 material.iOR = MEDIUM_IOR*leaving+material.iOR*(1.0-
leaving);\n\
                 \n\
                 // Shadow\n\
                 vec3 shLight = material.opacity*ray.light*\n\
```

```
                        (material.ambient+material.diffuse+\n\
                        material.specular);\n\
                  \n\
                  if(sum(shLight) > 0.0) {\n\
                     /* Get the hit color */\n\
                     vec3 ambient =
ambientLightColor*material.ambient;\n\
                     \n\
                     for(int l = 0; l < MAX_POINT_LIGHTS; ++l) {\n\
                        vec3 lightColor = pointLightColor[l];\n\
                        \n\
                        vec3 toLight = pointLightPosition[l]-hit;\n\
                        float dist = length(toLight);\n\
                        \n\
                        toLight /= dist;\n\
                        \n\
                        vec3 fromLight = -toLight;\n\
                        \n\
                        /* If falloff distance, determine
attenuation\n\

   http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                        float attenuation = 1.0,\n\
                           lightRange = pointLightDistance[l];\n\
                        \n\
                        if(lightRange > 0.0) {\n\
                           float distFactor = dist/\n\
                                 (1.0-pow(dist/lightRange, 2.0));\n\
                           \n\
                           attenuation = 1.0/pow(distFactor+1.0,
2.0);\n\
                        }\n\
                        \n\
                        float lightCosAngle = spotLightCosAngle[l];\n\
                        \n\
                        /* If spot light, determine if within spot
light cone */\n\
                        //if(attenuation > 0.0 && spotLightAngle[l] <=
180.0) {/* In the range [0-180], exclusive - should it be [0, 1]-
>[90, 0]? (see attenuation with spot falloff) */\n\
                        // float cosAngle = dot(fromLight,\n\
                        //    spotLightDirection[l]);\n\
                        // \n\
                        // /* Which will result in proper
attenuation?\n\
                        //    http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\
                        //
   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
```

```
                        // attenuation *= ceil(cosAngle-
lightCosAngle)*\n\
                        //    pow(cosAngle, spotLightFalloff[l]);\n\
                        //    //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        //}\n\
                        float cosAngle = dot(fromLight,
spotLightDirection[l]);\n\
                        \n\
                        attenuation *= ceil(attenuation)*ceil(180.0-
spotLightAngle[l])*\n\
                           ceil(cosAngle-lightCosAngle)*pow(cosAngle,
spotLightFalloff[l]);\n\
                           //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        \n\
                        if(attenuation > 0.0) {   /* Don\'t bother
calculating color or casting if outside of light range */\n\
                           /* Compute the phong shading at the hit\n\
                              Ambient is only accumulated once, so
do\n\
                              that outside this loop */\n\
                           vec3 rayColor;\n\
                           \n\
                           vec3 diffuse =
material.diffuse*lightColor*\n\
                                 max(dot(hitNormal, toLight), 0.0);\n\
                           \n\
                           /* TODO: texturing *//*\n\
                           if(material.textureID >= 0) {\n\
                              vec4 texel =
texture2D(textures[material.textureID],\n\
                                 getTextureCoord(meshID, hit));\n\
                              \n\
                              diffuse *= texel.rgb*texel.a;\n\
                           }*/\n\
                           \n\
                           rayColor += diffuse;\n\
                           \n\
                           vec3 reflection = reflect(toLight,
hitNormal);\n\
                           float highlight = max(dot(reflection,
ray.dir), 0.0);\n\
                           vec3 specular =
material.specular*lightColor*\n\
                                 pow(highlight, material.shine);\n\
                           \n\
                           rayColor += specular;\n\
                           rayColor *= material.opacity;\n\
                           \n\
```

```
                        if(sum(rayColor) > 0.0) { /* Don\'t bother
casting if it\'s just black anyway  */\n\
                           /* TODO: take jittered samples for soft
shadows? */\n\
                           ray = Ray(hit, toLight, h,\n\
                              shLight*rayColor*attenuation,\n\
                              material.iOR, dist);\n\
                           \n\
                           t = intersection(ray, meshID, triangle,
hit);\n\
                           \n\
                           /* Calculate color from closest
intersection */\n\
                           if(EPSILON <= ray.tLight &&\n\
                              (ray.tLight <= t || t < EPSILON)) {
   /* Light closer or no hit */\n\
                              accColor += ray.light;\n\
                           }\n\
                        }\n\
                     }\n\
                  }\n\
                  \n\
                  accColor += shLight*ambient*material.opacity;\n\
               }\n\
            }\n\
            \n\
            return accColor;\n\
         }',

      fragmentMain:
         'void main() {\n\
            gl_FragColor = vec4(traceRays(Ray(cameraPosition,\n\
                  normalize(pos.xyz-cameraPosition),\n\
                  0, vec3(1.0), MEDIUM_IOR, -1.0)), 1.0);\n\n'+
            'gl_FragColor += vec4(pos.z/100.0);\n\n'+

            THREE.ShaderChunk["fog_fragment"]+'\n\n'+
            /* TODO: alter traceRays to write the meshID of the
               first intersection into a variable, which can then
               be assigned to the alpha part of the fragment color
               This could then be read back on the CPU
               Finally, the FBO texture could be rendered out to
               the screen with all alpha set to 1.0 in a post pass
               void readPixels(GLint x, GLint y, GLsizei width,
GLsizei height, GLenum format, GLenum type, ArrayBufferView? pixels)
*/
            '}'
      });
```

B

```
function RayTracer(options) {
        if(!options) { options = {}; }

        // Note: float texture values go from [-inf, +inf], they are
NOT normalised to [0, 1] -
http://stackoverflow.com/questions/5709023/what-exactly-is-a-
floating-point-texture
        this.uniforms = THREE.UniformsUtils.merge([
           THREE.UniformsLib.fog,
           THREE.UniformsLib.particle,
           {
              ambientLightColor: { type: 'fv', value: [] },

              directionalLightColor: { type: 'fv', value: [] },
              directionalLightDirection: { type: 'fv', value: [] },

              pointLightColor: { type: 'fv', value: [] },
              pointLightPosition: { type: 'fv', value: [] },
              pointLightDistance: { type: 'fv1', value: [] },

              spotLightDirection: { type: 'fv', value: [] },
              spotLightCosAngle: { type: 'fv1', value: [] },
              spotLightAngle: { type: 'fv1', value: [] },
              spotLightFalloff: { type: 'fv1', value: [] },

              /* M*(1*RGB+4*RGBA - [xpos, ypos, zpos, radius]
                    [reflect, iOR, density, opacity]
                    [ramb, gamb, bamb]
                    [rdiff, gdiff, bdiff, textureID]
                    [rspec, gspec, bspec, shine] */
              meshes: { type: 't', value: 1,
                 texture: new THREE.DataTexture(new Float32Array(0),
                    0, 0, THREE.RGBAFormat, THREE.FloatType,
                    undefined, undefined, undefined,
                    THREE.NearestFilter, THREE.NearestFilter) },

              numMeshes: { type: 'f', value: 0 },

              infinity: { type: 'f', value: (options.infinity ||
10000) },

              maxHits: { type: 'i', value: 0 },
              maxRays: { type: 'i', value: 0 },

              // TODO: better...
              backdropID: { type: 'f', value: 0 }
           }]);

        this.material = new THREE.ShaderMaterial({
```

```
            fragmentShader: '', vertexShader: this.vertex,
            uniforms: this.uniforms, lights: true, fog: true
         });

         this.scene = options.scene;
         this.entities = options.entities;

         var b = options.budgets, growth = 1.5;
         this.budgets = {
            autoSetup: false,
            growth: growth,

            lights: this.scene.lights.length,
            meshes: (($.isNumeric(b.meshes))?
               b.meshes : Math.floor(this.entities.length*growth)),
            hit: 0, ray: 0, quadTree: 0
         };

         this.setHitBudget(b.hit || 3)
            .setHitMax(options.maxHits || 3)
            .update().setup();

         this.budgets.autoSetup = !!b.autoSetup;
      }
      $.extend(RayTracer.prototype, {
         setup: function() {
            var b = this.budgets;

            this.material.fragmentShader =
               '#define HIT_BUDGET '+b.hit+'\n\
               #define HIT_BUDGET_F '+b.hit+'.0\n\
               #define RAY_BUDGET '+b.ray+'\n\
               #define RAY_BUDGET_F '+b.ray+'.0\n\
               #define MESHES_BUDGET '+b.meshes+'\n\
               #define MESHES_BUDGET_F '+b.meshes+'.0\n\n'+

               this.fragmentHeader+'\n'+this.structs+'\n'+this.util+
               '\n'+this.readData+'\n'+this.intersections+'\n'+
               this.intersectAllRadii+'\n'+

   //this.intersectRadii+'\n'+this.simpleIntersectSceneFlat+'\n'+

   this.simpleShadowRayTracerOneLight+'\n'+this.fragmentMain;

            // Dirty material(?)
            this.refresh();

            return this;
         },
         update: function() { return
this.updateDataTextures().updateLights(); },
```

```
        updateDataTextures: function() {
           var u = this.uniforms,
              meshesTexture = u.meshes.texture,
              meshesData = [],
              eL = this.entities.length;

           // Mesh tier
           for(var e = 0; e < eL; ++e) {
              var mesh = this.entities[e],
                 shape = mesh.shape,
                 bRad = shape.boundRad,

                 mat = shape.material,
                 diff = mat.diffuse,
                 amb = mat.ambient,
                 spec = mat.specular;

              meshesData = meshesData.concat(
                 bRad.pos.x, bRad.pos.y, shape.three.position.z,
bRad.rad,
                 mat.reflect, mat.iOR, mat.density, mat.opacity,
                 amb.r, amb.g, amb.b, 0,
                 diff.r, diff.g, diff.b, mat.textureID,
                 spec.r, spec.g, spec.b, mat.shine);

              if(mesh instanceof Lumens)
{ this.uniforms.backdropID.value = e; }
           }

           meshesTexture.image.height = u.numMeshes.value = eL;
           meshesTexture.image.width = ((eL)? 5 : 0);
           meshesTexture.image.data = new Float32Array(meshesData);
           meshesTexture.needsUpdate = true;

           var bud = this.budgets;

           if(bud.autoSetup) {
              if(eL > bud.meshes) {
                 bud.meshes = Math.floor(eL*bud.growth);
                 this.setup();
              }
           }

           return this;
        },
        updateLights: function() {
           /* Note: updating the number and types of lights can't be
done without
              rebuilding the shader program - set
material.needsUpdate to true
```

```
            (can't find this in the source though - delete
material.program instead?)
            https://github.com/mrdoob/three.js/wiki/Updates

            An alternative may be to set it up a budgeted number
of lights with
            dummy information (0 intensity)

            Then again, this could be out of date entirey... */
         var u = this.uniforms,
            d = u.spotLightDirection.value,
            ca = u.spotLightCosAngle.value,
            a = u.spotLightAngle.value,
            f = u.spotLightFalloff.value,

            lights = this.scene.lights, ll = lights.length;

         for(var l = 0, vecOffset = 0; l < ll; ++l) {
            var light = lights[l];

            if(light instanceof SpotLight) {
               d[vecOffset++] = light.direction.x;
               d[vecOffset++] = light.direction.y;
               d[vecOffset++] = light.direction.z;

               ca[l] = light.cosAngle;
               a[l] = Math.degrees(light.angle);
               f[l] = light.falloff;
            }
            else {
              d[vecOffset++] = d[vecOffset++] = d[vecOffset++] = 0;
               ca[l] = a[l] = f[l] = -1;
            }
         }

         if(this.budgets.autoSetup && ll > this.budgets.lights) {
            this.budgets.lights = ll;
            this.setup();
         }

         return this;
      },
      setHitBudget: function(hits) {
         this.budgets.hit = hits;
         this.budgets.ray = this.numRays(hits);

         return this;
      },
      setHitMax: function(hits) {
         this.uniforms.maxHits.value = hits;
         this.uniforms.maxRays.value = this.numRays(hits);
```

```
            if(this.budgets.autoSetup && hits > this.budgets.hit) {
               this.budgets.lights = this.scene.lights.length;
               this.setHitBudget(hits).setup();
            }

            return this;
         },
         numRays: function(hits) {
            // TODO: reflect and refract shadow rays for caustics etc
            /*return 1+(2+this.scene.lights.length)*
                  (Math.pow(2, hits)-1);*/
            return 2;
         },
         refresh: function() {
            //this.material.needsUpdate = true;  // r49+
            delete this.material.program;

            return this;
         },

         /* Supplied by THREE:
            uniform mat4 objectMatrix;
            uniform mat4 modelViewMatrix;
            uniform mat4 projectionMatrix;
            uniform mat4 viewMatrix;
            uniform mat3 normalMatrix;
            uniform vec3 cameraPosition;

            attribute vec3 position;
            attribute vec3 normal;
            attribute vec2 uv;
            attribute vec2 uv2;

            as well as color, morph targets, and skinning

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Renderer.js#L5273

            Set material.shading to THREE.SmoothShading to get
            per-vertex normals:

   https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Renderer.js#L774 */
         vertex:
            // For passing interpolated position to fragment
            'uniform float size;\n\
            varying vec3 pos;\n\
            \n\
            void main() {\n\
```

```
            //pos =
(projectionMatrix*modelViewMatrix*vec4(position, 1.0)).xyz;\n\
            pos = (objectMatrix*vec4(position, 1.0)).xyz;\n\
            gl_Position = projectionMatrix*modelViewMatrix*\n\
                  vec4(position, 1.0);\n\
         }',

      /* Supplied by THREE:
         uniform mat4 viewMatrix;
         uniform vec3 cameraPosition;

         as well as precision

  https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Renderer.js#L5362

         Set material.lights to true to use lights and include the
         corresponding variables:

  https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Shaders.js#L419,

  https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGL
Shaders.js#L458 */
      fragmentHeader:
         THREE.ShaderChunk["fog_pars_fragment"]+'\n\n'+
         /* TODO: use uniforms for eye ray */

         // toExponential ensures a float representation
         '#define PAD 0.25\n\
         \n\
         #define MESH_DATA_STEP '+(1.0/5.0).toExponential()+'\n\
         #define MESH_DATA_PAD '+(0.25/5.0).toExponential()+'\n\
         #define SPHERE_OFFSET 0.0\n\
         #define MATERIAL_OFFSET '+(1.0/5.0).toExponential()+'\n\
         \n\
         #define MEDIUM_IOR 1.0\n\
         #define EPSILON 0.01\n\
         \n\
         uniform vec3 ambientLightColor;\n\
         \n\
         #if MAX_POINT_LIGHTS > 0\n\
            uniform vec3 pointLightColor[MAX_POINT_LIGHTS];\n\
            uniform vec3 pointLightPosition[MAX_POINT_LIGHTS];\n\
            uniform float pointLightDistance[MAX_POINT_LIGHTS];\n\
            \n\
            uniform vec3 spotLightDirection[MAX_POINT_LIGHTS];\n\
            uniform float spotLightCosAngle[MAX_POINT_LIGHTS];\n\
            uniform float spotLightAngle[MAX_POINT_LIGHTS];\n\
            uniform float spotLightFalloff[MAX_POINT_LIGHTS];\n\
         #endif\n\
```

```
            #if MAX_DIR_LIGHTS > 0\n\
               uniform vec3 directionalLightColor[MAX_DIR_LIGHTS];\n\
               uniform vec3
directionalLightDirection[MAX_DIR_LIGHTS];\n\
            #endif\n\
            \n\
            /* Global scene data stored in textures */\n\
            uniform sampler2D meshes;\n\
            \n\
            uniform float numMeshes;\n\
            \n\
            uniform float infinity;\n\
            \n\
            uniform int maxHits;\n\
            uniform int maxRays;\n\
            \n\
            // TODO: better...\n\
            uniform float backdropID;\n\
            \n\
            varying vec3 pos;',

         structs:
            'struct Ray {\n\
               vec3 origin;\n\
               vec3 dir;\n\
               \n\
               int hit; /* Records the number of times this ray has
bounced in the scene */\n\
               \n\
               vec3 light; /* The light being transmitted (color,
energy) */\n\
               float iOR; /* The index of refraction of the medium
the ray is currently travelling through */\n\
               float tLight; /* The distance to the target light -
set for shadow rays, -1.0 otherwise */\n\
            };\n\
            \n\
            struct Sphere {\n\
               vec3 pos;\n\
               float rad;\n\
            };\n\
            \n\
            struct Material {\n\
               float reflect;\n\
               float iOR;\n\
               float density;\n\
               float opacity;\n\
               \n\
               vec3 ambient;\n\
               \n\
```

```
            vec3 diffuse; /* If textureID is valid, multiply texel
by diffuse, if not, just use diffuse */\n\
            int textureID;\n\
            \n\
            vec3 specular;\n\
            float shine;\n\
         };',

      util:
         'float sum(vec3 v) { return v.x+v.y+v.z; }\n\
         \n\
         bool inRange(float t) { return (EPSILON <= t && t <
infinity); }',

      /* Functions for reading in data from the respective
textures
         and returning a useable object */
      readData:
         'Sphere getSphere(float index) {\n\
            vec4 data = texture2D(meshes,\n\
                  vec2(SPHERE_OFFSET+MESH_DATA_PAD,\n\
                     (index+PAD)/numMeshes));\n\
            \n\
            return Sphere(data.xyz, data.w);\n\
         }\n\
         \n\
         Material getMaterial(float index) {\n\
            float i = MATERIAL_OFFSET+MESH_DATA_PAD,\n\
               j = (index+PAD)/numMeshes;\n\
            \n\
            vec4 data[4];\n\
            \n\
            data[0] = texture2D(meshes, vec2(i, j));\n\
            data[1] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
            data[2] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
            data[3] = texture2D(meshes, vec2(i += MESH_DATA_STEP,
j));\n\
            \n\
            return Material(data[0].r, data[0].g, data[0].b,
data[0].a,\n\
               data[1].rgb, data[2].rgb, int(data[2].a),\n\
               data[3].rgb, data[3].a);\n\
         }',

      /* The returned float (t) denotes the distance along
         the ray at which the intersection occurs, where valid
values
         for t are in the range [epsilon, infinity], exclusively
*/
```

```
        intersections:
           '/* See "Intersecting a Sphere" at
http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html */\n\
           float intersection(Ray ray, Sphere s) {\n\
              vec3 oc = s.pos-ray.origin;\n\
              float v = dot(oc, ray.dir);\n\
              float dSq = s.rad*s.rad-(dot(oc, oc)-v*v);\n\
              \n\
              return ((dSq >= 0.0)? v-sqrt(dSq) : -1.0);\n\
           }\n\
           \n\
           /* Ray-plane test */\n\
           float intersection(Ray ray, vec3 point, vec3 normal) {\n\
              float d = dot(ray.dir, normal);\n\
              \n\
              // Less than or equal to zero if parrallel or behind
ray\n\
              return ((abs(d) >= EPSILON)?\n\
                 dot(normal, (point-ray.origin))/d : -1.0);\n\
           }',

        intersectAllRadii:
           'float intersection(Ray ray, out float meshID, out vec3
hit, out vec3 hitNormal) {\n\
              float closest = infinity;\n\
              meshID = -1.0;\n\
              hit = vec3(0.0);\n\
              hitNormal = vec3(0.0);\n\
              \n\
              // Backdrop\n\
              vec3 pN = vec3(0.0, 0.0, 1.0);\n\
              float tB = intersection(ray, vec3(0.0), pN);\n\
              if(EPSILON <= tB && tB < closest) {\n\
                 closest = tB;\n\
                 meshID = backdropID;\n\
                 hit = ray.origin+ray.dir*tB;\n\
                 hitNormal = pN;\n\
              }\n\
              \n\
              for(float m = 0.0; m < MESHES_BUDGET_F; ++m) {\n\
                 if(m < numMeshes) {\n\
                    Sphere sphere = getSphere(m);\n\
                    float tR = intersection(ray, sphere);\n\
                    \n\
                    if(EPSILON <= tR && tR < closest) {\n\
                       closest = tR;\n\
                       meshID = m;\n\
                       hit = ray.origin+ray.dir*tR;\n\
                       hitNormal = normalize(hit-sphere.pos);\n\
                    }\n\
                    \n\
```

```
                    // float closer = ceil(closest-tR)*ceil(tR-
EPSILON);\n\
                    // float farther = (1.0-closer);\n\
                    // \n\
                    // closest = tR*closer+closest*farther;\n\
                    // meshID = m*closer+meshID*farther;\n\
                    // hit =
(ray.origin+ray.dir*tR)*closer+hit*farther;\n\
                    // hitNormal = normalize(hit-sphere.pos);\n\
                 }\n\
                 else { break; }  /* Check for end of list */\n\
              }\n\
              \n\
              return closest;\n\
           }',

        intersectRadii:
           'float intersection(Ray ray, float meshesIndex, float
meshCount, out float meshID, out vec3 hit, out vec3 hitNormal) {\n\
              float closest = infinity;\n\
              meshID = -1.0;\n\
              hit = vec3(0.0);\n\
              hitNormal = vec3(0.0);\n\
              \n\
              float mID = meshesIndex;\n\
              \n\
              for(float m = 0.0; m < MESHES_BUDGET_F; ++m) {\n\
                 if(m < meshCount && mID < numMeshes) {\n\
                    Sphere sphere = getSphere(mID);\n\
                    float tR = intersection(ray, sphere);\n\
                    \n\
                    if(EPSILON <= tR && tR < closest) {\n\
                       closest = tR;\n\
                       meshID = m;\n\
                       hit = ray.origin+ray.dir*tR;\n\
                       hitNormal = normalize(hit-sphere.pos);\n\
                    }\n\
                    \n\
                    // float closer = ceil(closest-tR)*ceil(tR-
EPSILON);\n\
                    // float farther = (1.0-closer);\n\
                    // \n\
                    // closest = tR*closer+closest*farther;\n\
                    // meshID = m*closer+meshID*farther;\n\
                    // hit =
(ray.origin+ray.dir*tR)*closer+hit*farther;\n\
                    // hitNormal = normalize(hit-sphere.pos);\n\
                    \n\
                    ++mID;\n\
                 }\n\
                 else { break; }  /* Check for end of list */\n\
```

```
            }\n\
            \n\
            return closest;\n\
         }',

      intersectScene:
         'float intersection(Ray ray, out float meshID, out vec3
hit) {\n\
            float closest = infinity;\n\
            meshID = -1.0;\n\
            hit = vec3(0.0);\n\
            \n\
            float nodes[QUAD_TREE_BUDGET];\n\
            nodes[0] = 0.0;\n\
            int nLast = 0;\n\
            \n\
            /* Traverse quadTree */\n\
            for(int n = 0; n < QUAD_TREE_BUDGET; ++n) {\n\
               if(n <= nLast && float(n) < numNodes) {\n\
                  Node node = getNode(nodes[n]);\n\
                  bool intersects = inRange(intersection(ray,
node.bounds));\n\
                  \n\
                  if(intersects && node.numSubNodes > 0.0) {\n\
                     for(int nl = 0; nl < QUAD_TREE_BUDGET; ++nl)
{\n\
                        if(nl > nLast) { /* Super-ugly hack to get
the last element in the flat node list, since GLSL ES doesn\'t allow
non-constant expressions to be used to index an array (so no
"nodes[++nLast] = node.subNodesIndex", which is the nice way to do
it) */\n\
                           nodes[nl] = node.subNodesIndex;\n\
                           nodes[nl+1] = node.subNodesIndex+1.0;\n\
                           nodes[nl+2] = node.subNodesIndex+2.0;\n\
                           nodes[nl+3] = node.subNodesIndex+3.0;\n\
                           \n\
                           nodes[nl+4] = -1.0;\n\
                           nLast += 4;\n\
                           break;\n\
                        }\n\
                     }\n\
                  }\n\
                  \n\
                  if(intersects && node.meshCount > 0.0) {   /*
Check at every level to avoid rechecking borderKids by passing them
down to each child node */\n\
                     float mID;\n\
                     vec3 h;\n\
                     float mT = intersection(ray,
node.meshesIndex,\n\
                           node.meshCount, mID, h);\n\
```

```
                        \n\
                        if(mT < closest) {\n\
                           closest = mT;\n\
                           meshID = mID;\n\
                           hit = h;\n\
                        }\n\
                     }\n\
                  }\n\
                  else { break; }\n\
               }\n\
               \n\
               return closest;\n\
            }',

         intersectSceneFlat:
            '/* Finds the closest intersection along the ray with
the\n\
               entire scene, in object space */\n\
            float intersection(Ray ray, out float meshID,\n\
               out Triangle triangle, out vec3 hit) {\n\
               float closest = infinity;\n\
               Triangle dummy;\n\
               triangle = dummy;\n\
               meshID = -1.0;\n\
               hit = vec3(0.0);\n\
               \n\
               /* Traverse quadTree */\n\
               for(float n = 0.0; n < QUAD_TREE_BUDGET_F; ++n) {\n\
                  if(n < numNodes) {\n\
                     Node node = getNode(n);\n\
                     \n\
                     if(node.meshCount > 0.0 &&
inRange(intersection(ray, node.bounds))) {   /* Check at every level
to avoid rechecking borderKids by passing them down to each child
node */\n\
                        float mID;\n\
                        Triangle tri;\n\
                        vec3 h;\n\
                        float mT = intersection(ray,
node.meshesIndex,\n\
                              node.meshCount, mID, tri, h);\n\
                        \n\
                        if(mT < closest) {\n\
                           closest = mT;\n\
                           triangle = tri;\n\
                           meshID = mID;\n\
                           hit = h;\n\
                        }\n\
                     }\n\
                  }\n\
                  else { break; }\n\
```

```
            }\n\
            \n\
            return closest;\n\
         }',

      simpleIntersectScene:
         'float intersection(Ray ray, out float meshID, out vec3
hit, out vec3 hitNormal) {\n\
            float closest = infinity;\n\
            meshID = -1.0;\n\
            hit = vec3(0.0);\n\
            hitNormal = vec3(0.0);\n\
            \n\
            float nodes[QUAD_TREE_BUDGET];\n\
            nodes[0] = 0.0;\n\
            int nLast = 0;\n\
            \n\
            /* Traverse quadTree */\n\
            for(int n = 0; n < QUAD_TREE_BUDGET; ++n) {\n\
               if(n <= nLast && float(n) < numNodes) {\n\
                  Node node = getNode(nodes[n]);\n\
                  bool intersects = inRange(intersection(ray,
node.bounds));\n\
                  \n\
                  if(intersects && node.numSubNodes > 0.0) {\n\
                     for(int nl = 0; nl < QUAD_TREE_BUDGET; ++nl)
{\n\
                        if(nl > nLast) { /* Super-ugly hack to get
the last element in the flat node list, since GLSL ES doesn\'t allow
non-constant expressions to be used to index an array (so no
"nodes[++nLast] = node.subNodesIndex", which is the nice way to do
it) */\n\
                           nodes[nl] = node.subNodesIndex;\n\
                           nodes[nl+1] = node.subNodesIndex+1.0;\n\
                           nodes[nl+2] = node.subNodesIndex+2.0;\n\
                           nodes[nl+3] = node.subNodesIndex+3.0;\n\
                           \n\
                           nodes[nl+4] = -1.0;\n\
                           nLast += 4;\n\
                           break;\n\
                        }\n\
                     }\n\
                  }\n\
                  \n\
                  if(intersects && node.meshCount > 0.0) {  /*
Check at every level to avoid rechecking borderKids by passing them
down to each child node */\n\
                     float mID;\n\
                     vec3 h;\n\
                     vec3 hn;\n\
```

```
                        float mT = intersection(ray,
node.meshesIndex,\n\
                              node.meshCount, mID, h, hn);\n\
                        \n\
                        if(mT < closest) {\n\
                           closest = mT;\n\
                           meshID = mID;\n\
                           hit = h;\n\
                           hitNormal = hn;\n\
                        }\n\
                     }\n\
                  }\n\
                  else { break; }\n\
               }\n\
               \n\
               return closest;\n\
            }',

         simpleIntersectSceneFlat:
            '/* Finds the closest intersection along the ray with
the\n\
               entire scene, in object space */\n\
            float intersection(Ray ray, out float meshID, out vec3
hit, out vec3 hitNormal) {\n\
               float closest = infinity;\n\
               meshID = -1.0;\n\
               hit = vec3(0.0);\n\
               hitNormal = vec3(0.0);\n\
               \n\
               /* Traverse quadTree */\n\
               for(float n = 0.0; n < QUAD_TREE_BUDGET_F; ++n) {\n\
                  if(n < numNodes) {\n\
                     Node node = getNode(n);\n\
                     \n\
                     if(node.meshCount > 0.0 &&
inRange(intersection(ray, node.bounds))) {   /* Check at every level
to avoid rechecking borderKids by passing them down to each child
node */\n\
                        float mID;\n\
                        vec3 h;\n\
                        vec3 hn;\n\
                        float mT = intersection(ray,
node.meshesIndex,\n\
                              node.meshCount, mID, h, hn);\n\
                        \n\
                        if(mT < closest) {\n\
                           closest = mT;\n\
                           meshID = mID;\n\
                           hit = h;\n\
                           hitNormal = hn;\n\
                        }\n\
```

```
                  }\n\
               }\n\
               else { break; }\n\
            }\n\
            \n\
            return closest;\n\
         }',

      /* Given an initial ray, traces rays around the
         scene up to the maximum number allowed
         (maintaining recursive heirarchical order),
         and returns the accumulated color of the fragment */
      whittedRayTracer:
         'vec3 traceRays(Ray eyeRay) {\n\
            vec3 accColor;\n\
            \n\
            Ray rays[RAY_BUDGET];\n\
            \n\
            /* TODO: figure out if the eye ray needs to be
traced,\n\
               or if nothing is between the fragment and the\n\
               camera, since it is being drawn */\n\
            rays[0] = eyeRay;\n\
            \n\
            /* rLast - for reducing the number of loops\n\
               When spawning rays, add them to rays after this\n\
               and increase it by the number of added rays\n\
               If r exceeds this, break (no more valid rays to
be\n\
               traced) */\n\
            int rLast = 0;\n\
            \n\
            float meshID;\n\
            Triangle tri;\n\
            vec3 hit;\n\
            \n\
            for(int r = 0; r < RAY_BUDGET; ++r) {\n\
               if(r <= rLast && r < maxRays) {\n\
                  Ray ray = rays[r];\n\
                  float t = intersection(ray, meshID, tri, hit);\n\
                  \n\
                  /* Calculate color from closest intersection
*/\n\
                  if(EPSILON <= ray.tLight && /* Shadow ray */\n\
                     (ray.tLight <= t || t < EPSILON)) {  /* Light
closer or no hit */\n\
                     /* TODO: caustics, reflected light\n\
                        Cast further reflected and refracted
shadow\n\
                        rays which accumulate color correctlty in
the\n\
```

```
                        case of an intersection with a
reflective\n\
                        or refractive mesh */\n\
                     /* Final color accumulation - direct light
*/\n\
                     accColor += ray.light;\n\
                  }\n\
                  else if(inRange(t) && ray.hit < maxHits) {\n\
                     /* Cast shadow, reflection, and\n\
                        refraction rays, if the energy is\n\
                        over the threshold */\n\
                     int h = ray.hit+1;\n\
                     \n\
                     mat4 objMat = getObjectMatrix(meshID);\n\
                     hit = (objMat*vec4(hit, 1.0)).xyz;\n\
                     Triangle triangle = transform(tri, objMat,
getNormalMatrix(meshID));\n\
                     \n\
                     /* TODO: get smooth normal -
http://www.codeproject.com/Articles/20144/Simple-Ray-Tracing-in-C-
Part-VI-Vertex-Normal-Inte */\n\
                     vec3 hitNormal = triangle.faceNormal;\n\
                     Material material = getMaterial(meshID);\n\
                     \n\
                     /* Set iOR according to whether leaving or\n\
                        entering material */\n\
                     /*if(dot(ray.dir, hitNormal) >= 0.0) {\n\
                        material.iOR = MEDIUM_IOR;\n\
                     }*/\n\
                     float leaving = ceil(dot(ray.dir,
hitNormal));\n\
                     material.iOR =
MEDIUM_IOR*leaving+material.iOR*(1.0-leaving);\n\
                     \n\
                     // Shadow\n\
                     vec3 shLight = material.opacity*ray.light*\n\
                           (material.ambient+material.diffuse+\n\
                           material.specular);\n\
                     \n\
                     if(sum(shLight) > 0.0) {\n\
                        /* Get the hit color */\n\
                        /* Branching is slow - only do it there\'s
a\n\
                           significant chunk of code inside */\n\
                        vec3 ambient =
ambientLightColor*material.ambient;\n\
                        \n\
                        for(int l = 0; l < MAX_POINT_LIGHTS; ++l)
{\n\
                           vec3 lightColor = pointLightColor[l];\n\
                           \n\
```

```
                              if(sum(lightColor) > 0.0) {\n\
                                 vec3 toLight = pointLightPosition[l]-
hit;\n\
                                 float dist = length(toLight);\n\
                                 \n\
                                 toLight /= dist;\n\
                                 \n\
                                 vec3 fromLight = -toLight;\n\
                                 \n\
                                 /* If falloff distance, determine
attenuation\n\

   http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                                 float attenuation = 1.0,\n\
                                    lightRange =
pointLightDistance[l];\n\
                                 \n\
                                 if(lightRange > 0.0) {\n\
                                    float distFactor = dist/\n\
                                          (1.0-pow(dist/lightRange,
2.0));\n\
                                    \n\
                                    attenuation =
1.0/pow(distFactor+1.0, 2.0);\n\
                                 }\n\
                                 \n\
                                 float lightCosAngle =
spotLightCosAngle[l];\n\
                                 \n\
                                 /* If spot light, determine if within
spot light cone */\n\
                                 if(attenuation > 0.0 &&\n\
                                    -1.0 < lightCosAngle &&
lightCosAngle < 1.0) { /* In the range [0-180], exclusive - should it
be [0, 1]->[90, 0]? (see attenuation with spot falloff) */\n\
                                    float cosAngle = dot(fromLight,\n\
                                       spotLightDirection[l]);\n\
                                    \n\
                                    attenuation *= ((lightCosAngle <=
cosAngle)?\n\
                                       /* Which will result in proper
attenuation?\n\
                                          http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\

   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
                                          pow(cosAngle,
spotLightFalloff[l]) : 0.0);\n\
```

```
                                    //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]) : 0.0);\n\
                                }\n\
                                \n\
                                if(attenuation > 0.0) {   /* Don\'t
bother calculating color or casting if outside of light range */\n\
                                   /* Compute the phong shading at the
hit\n\
                                     Ambient is only accumulated once,
so do\n\
                                     that outside this loop */\n\
                                   vec3 rayColor;\n\
                                   \n\
                                   vec3 diffuse =
material.diffuse*lightColor*\n\
                                      max(dot(hitNormal, toLight),
0.0);\n\
                                   \n\
                                   /* TODO: texturing *//*\n\
                                   if(material.textureID >= 0) {\n\
                                      vec4 texel =
texture2D(textures[material.textureID],\n\
                                         getTextureCoord(meshID,
hit));\n\
                                      \n\
                                      diffuse *= texel.rgb*texel.a;\n\
                                   }*/\n\
                                   \n\
                                   rayColor += diffuse;\n\
                                   \n\
                                   if(sum(material.specular) > 0.0)
{\n\
                                      vec3 reflection =
reflect(toLight, hitNormal);\n\
                                      float highlight =
max(dot(reflection, ray.dir), 0.0);\n\
                                      vec3 specular =
material.specular*lightColor*\n\
                                           pow(highlight,
material.shine);\n\
                                      \n\
                                      rayColor += specular;\n\
                                   }\n\
                                   \n\
                                   rayColor *= material.opacity;\n\
                                   \n\
                                   if(sum(rayColor) > 0.0) { /* Don\'t
bother casting if it\'s just black anyway  */\n\
                                      /* TODO: take jittered samples
for soft shadows? */\n\
                                      Ray shadow = Ray(hit,\n\
```

```
                                        toLight, h,\n\
                                    
   shLight*rayColor*attenuation,\n\
                                        material.iOR, dist);\n\
                                    \n\
                                    float meshIDSh;\n\
                                    Triangle triangleSh;\n\
                                    vec3 hitSh;\n\
                                    \n\
                                    float tSh =
intersection(shadow,\n\
                                        meshIDSh, triangleSh,
hitSh);\n\
                                    \n\
                                    /* Calculate color from closest
intersection */\n\
                                    if(EPSILON <= shadow.tLight
&&\n\
                                        (ray.tLight <= tSh || tSh <
EPSILON)) {/* Light closer or no hit */\n\
                                        accColor += shadow.light;\n\
                                    }\n\
                                }\n\
                            }\n\
                        }\n\
                    }\n\
                    \n\
                    accColor +=
shLight*ambient*material.opacity;\n\
                }\n\
                \n\
                // Reflection\n\
                vec3 rflLight =
material.reflect*material.opacity*ray.light;\n\
                if(sum(rflLight) > 0.0) {\n\
                  for(int rl = 0; rl < QUAD_TREE_BUDGET; ++rl)
{\n\
                        if(rl > rLast) {\n\
                            rays[rl] = Ray(hit,\n\
                                reflect(ray.dir, hitNormal), h,\n\
                                rflLight, material.iOR, -1.0);\n\
                            \n\
                            rLast = rl;\n\
                            break;\n\
                        }\n\
                    }\n\
                }\n\
                \n\
                // Refraction - TODO: Beer\'s Law -
http://www.flipcode.com/archives/Raytracing_Topics_Techniques-
Part_3_Refractions_and_Beers_Law.shtml\n\
```

```
                        vec3 rfrLight = (1.0-
material.opacity)*ray.light;\n\
                        if(sum(rfrLight) > 0.0) {\n\
                          for(int rl = 0; rl < QUAD_TREE_BUDGET; ++rl)
{\n\
                              if(rl > rLast) {\n\
                                 rays[rl] = Ray(hit,\n\
                                    refract(ray.dir, hitNormal,\n\
                                       ray.iOR/material.iOR), h,\n\
                                    rfrLight, material.iOR, -1.0);\n\
                                 \n\
                                 rLast = rl;\n\
                                 break;\n\
                              }\n\
                           }\n\
                        }\n\
                     }\n\
                  }\n\
                  else { break; }\n\
               }\n\
               \n\
               return accColor;\n\
            }',

         /* Linear ray tracer (doesn't branch into a tree),
            shadow rays */
         shadowRayTracer:
            'vec3 traceRays(Ray eyeRay) {\n\
               vec3 accColor;\n\
               Ray ray = eyeRay;\n\
               float meshID;\n\
               Triangle tri;\n\
               vec3 hit;\n\
               float t = intersection(ray, meshID, tri, hit); // TODO:
remove this intersection, pass each mesh\'s data in uniforms for
this instead \n\
               \n\
               if(inRange(t)) {\n\
                  /* Cast shadow ray, if the energy is over the
threshold */\n\
                  int h = ray.hit+1;\n\
                  \n\
                  mat4 objMat = getObjectMatrix(meshID);\n\
                  hit = (objMat*vec4(hit, 1.0)).xyz;\n\
                  Triangle triangle = transform(tri, objMat,
getNormalMatrix(meshID));\n\
                  \n\
                  /* TODO: get smooth normal */\n\
                  vec3 hitNormal = triangle.faceNormal;\n\
                  Material material = getMaterial(meshID);\n\
                  \n\
```

```
                float leaving = ceil(dot(ray.dir, hitNormal));\n\
                material.iOR = MEDIUM_IOR*leaving+material.iOR*(1.0-
leaving);\n\
                \n\
                // Shadow\n\
                vec3 shLight = material.opacity*ray.light*\n\
                     (material.ambient+material.diffuse+\n\
                     material.specular);\n\
                \n\
                if(sum(shLight) > 0.0) {\n\
                   /* Get the hit color */\n\
                   vec3 ambient =
ambientLightColor*material.ambient;\n\
                   \n\
                   for(int l = 0; l < MAX_POINT_LIGHTS; ++l) {\n\
                      vec3 lightColor = pointLightColor[l];\n\
                      \n\
                      vec3 toLight = pointLightPosition[l]-hit;\n\
                      float dist = length(toLight);\n\
                      \n\
                      toLight /= dist;\n\
                      \n\
                      vec3 fromLight = -toLight;\n\
                      \n\
                      /* If falloff distance, determine
attenuation\n\

  http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                      float attenuation = 1.0,\n\
                         lightRange = pointLightDistance[l];\n\
                      \n\
                      if(lightRange > 0.0) {\n\
                         float distFactor = dist/\n\
                               (1.0-pow(dist/lightRange, 2.0));\n\
                         \n\
                         attenuation = 1.0/pow(distFactor+1.0,
2.0);\n\
                      }\n\
                      \n\
                      float lightCosAngle = spotLightCosAngle[l];\n\
                      \n\
                      /* If spot light, determine if within spot
light cone */\n\
                      //if(attenuation > 0.0 && spotLightAngle[l] <=
180.0) {/* In the range [0-180], exclusive - should it be [0, 1]-
>[90, 0]? (see attenuation with spot falloff) */\n\
                      // float cosAngle = dot(fromLight,\n\
                      //    spotLightDirection[l]);\n\
                      // \n\
```

```
                        // /* Which will result in proper
attenuation?\n\
                        //    http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\
                        //
   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
                        // attenuation *= ceil(cosAngle-
lightCosAngle)*\n\
                        //    pow(cosAngle, spotLightFalloff[l]);\n\
                        //    //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        //}\n\
                        float cosAngle = dot(fromLight,
spotLightDirection[l]);\n\
                        \n\
                        attenuation *= ceil(attenuation)*ceil(180.0-
spotLightAngle[l])*\n\
                           ceil(cosAngle-lightCosAngle)*pow(cosAngle,
spotLightFalloff[l]);\n\
                           //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        \n\
                        if(attenuation > 0.0) {   /* Don\'t bother
calculating color or casting if outside of light range */\n\
                           /* Compute the phong shading at the hit\n\
                              Ambient is only accumulated once, so
do\n\
                              that outside this loop */\n\
                           vec3 rayColor;\n\
                           \n\
                           vec3 diffuse =
material.diffuse*lightColor*\n\
                                 max(dot(hitNormal, toLight), 0.0);\n\
                           \n\
                           /* TODO: texturing *//*\n\
                           if(material.textureID >= 0) {\n\
                              vec4 texel =
texture2D(textures[material.textureID],\n\
                                 getTextureCoord(meshID, hit));\n\
                              \n\
                              diffuse *= texel.rgb*texel.a;\n\
                           }*/\n\
                           \n\
                           rayColor += diffuse;\n\
                           \n\
                           vec3 reflection = reflect(toLight,
hitNormal);\n\
                           float highlight = max(dot(reflection,
ray.dir), 0.0);\n\
```

```
                           vec3 specular =
material.specular*lightColor*\n\
                                 pow(highlight, material.shine);\n\
                           \n\
                           rayColor += specular;\n\
                           rayColor *= material.opacity;\n\
                           \n\
                           if(sum(rayColor) > 0.0) { /* Don\'t bother
casting if it\'s just black anyway  */\n\
                              /* TODO: take jittered samples for soft
shadows? */\n\
                              ray = Ray(hit, toLight, h,\n\
                                 shLight*rayColor*attenuation,\n\
                                 material.iOR, dist);\n\
                              \n\
                              t = intersection(ray, meshID, triangle,
hit);\n\
                              \n\
                              /* Calculate color from closest
intersection */\n\
                              if(EPSILON <= ray.tLight &&\n\
                                 (ray.tLight <= t || t < EPSILON)) {
   /* Light closer or no hit */\n\
                                 accColor += ray.light;\n\
                              }\n\
                           }\n\
                        }\n\
                     }\n\
                     \n\
                     accColor += shLight*ambient*material.opacity;\n\
                  }\n\
               }\n\
               \n\
               return accColor;\n\
            }',

         simpleShadowRayTracer:
            'vec3 traceRays(Ray eyeRay) {\n\
               vec3 accColor;\n\
               Ray ray = eyeRay;\n\
               float meshID;\n\
               vec3 hit;\n\
               float t = intersection(ray, meshID, hit); // TODO:
remove this intersection, pass each mesh\'s data in uniforms for
this instead \n\
               \n\
               if(inRange(t)) {\n\
                  /* Cast shadow ray, if the energy is over the
threshold */\n\
                  int h = ray.hit+1;\n\
                  \n\
```

```
                mat4 objMat = getObjectMatrix(meshID);\n\
                hit = (objMat*vec4(hit, 1.0)).xyz;\n\
                vec3 center = (objMat*vec4(vec3(0.0), 1.0)).xyz;\n\
                vec3 hitNormal = normalize(hit-center);\n\
                Material material = getMaterial(meshID);\n\
                \n\
                float leaving = ceil(dot(ray.dir, hitNormal));\n\
                material.iOR = MEDIUM_IOR*leaving+material.iOR*(1.0-
leaving);\n\
                \n\
                // Shadow\n\
                vec3 shLight = material.opacity*ray.light*\n\
                     (material.ambient+material.diffuse+\n\
                     material.specular);\n\
                \n\
                if(sum(shLight) > 0.0) {\n\
                   /* Get the hit color */\n\
                   vec3 ambient =
ambientLightColor*material.ambient;\n\
                   \n\
                   for(int l = 0; l < MAX_POINT_LIGHTS; ++l) {\n\
                      vec3 lightColor = pointLightColor[l];\n\
                      \n\
                      vec3 toLight = pointLightPosition[l]-hit;\n\
                      float dist = length(toLight);\n\
                      \n\
                      toLight /= dist;\n\
                      \n\
                      vec3 fromLight = -toLight;\n\
                      \n\
                      /* If falloff distance, determine
attenuation\n\

   http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                      float attenuation = 1.0,\n\
                         lightRange = pointLightDistance[l];\n\
                      \n\
                      if(lightRange > 0.0) {\n\
                         float distFactor = dist/\n\
                               (1.0-pow(dist/lightRange, 2.0));\n\
                         \n\
                         attenuation = 1.0/pow(distFactor+1.0,
2.0);\n\
                      }\n\
                      \n\
                      float lightCosAngle = spotLightCosAngle[l];\n\
                      \n\
                      /* If spot light, determine if within spot
light cone */\n\
```

```
                        //if(attenuation > 0.0 && spotLightAngle[l] <=
180.0) {/* In the range [0-180], exclusive - should it be [0, 1]-
>[90, 0]? (see attenuation with spot falloff) */\n\
                        // float cosAngle = dot(fromLight,\n\
                        //    spotLightDirection[l]);\n\
                        // \n\
                        // /* Which will result in proper
attenuation?\n\
                        //    http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\
                        //
   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
                        // attenuation *= ceil(cosAngle-
lightCosAngle)*\n\
                        //    pow(cosAngle, spotLightFalloff[l]);\n\
                        //    //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        //}\n\
                        float cosAngle = dot(fromLight,
spotLightDirection[l]);\n\
                        \n\
                        attenuation *= ceil(attenuation)*ceil(180.0-
spotLightAngle[l])*\n\
                           ceil(cosAngle-lightCosAngle)*pow(cosAngle,
spotLightFalloff[l]);\n\
                              //pow((cosAngle+1.0)*0.5,
spotLightFalloff[l]);\n\
                        \n\
                        if(attenuation > 0.0) {   /* Don\'t bother
calculating color or casting if outside of light range */\n\
                           /* Compute the phong shading at the hit\n\
                              Ambient is only accumulated once, so
do\n\
                              that outside this loop */\n\
                           vec3 rayColor;\n\
                           \n\
                           vec3 diffuse =
material.diffuse*lightColor*\n\
                                 max(dot(hitNormal, toLight), 0.0);\n\
                           \n\
                           /* TODO: texturing *//*\n\
                           if(material.textureID >= 0) {\n\
                              vec4 texel =
texture2D(textures[material.textureID],\n\
                                 getTextureCoord(meshID, hit));\n\
                              \n\
                              diffuse *= texel.rgb*texel.a;\n\
                           }*/\n\
                           \n\
                           rayColor += diffuse;\n\
```

```
                        \n\
                        vec3 reflection = reflect(toLight,
hitNormal);\n\
                        float highlight = max(dot(reflection,
ray.dir), 0.0);\n\
                        vec3 specular =
material.specular*lightColor*\n\
                              pow(highlight, material.shine);\n\
                        \n\
                        rayColor += specular;\n\
                        rayColor *= material.opacity;\n\
                        \n\
                        if(sum(rayColor) > 0.0) {/* Don\'t bother
casting if it\'s just black anyway  */\n\
                           /* TODO: take jittered samples for soft
shadows? */\n\
                           ray = Ray(hit, toLight, h,\n\
                              shLight*rayColor*attenuation,\n\
                              material.iOR, dist);\n\
                           \n\
                           t = intersection(ray, meshID, hit);\n\
                           \n\
                           /* Calculate color from closest
intersection */\n\
                           /*if(EPSILON <= ray.tLight &&\n\
                              (ray.tLight <= t || t < EPSILON)) {
   // Light closer or no hit\n\
                              accColor += ray.light;\n\
                           }*/\n\
                           \n\
                           accColor += ray.light*(ceil(t-
ray.tLight)+ceil(EPSILON-t));\n\
                        }\n\
                     }\n\
                  }\n\
                  \n\
                  accColor += shLight*ambient*material.opacity;\n\
               }\n\
            }\n\
            \n\
            return accColor;\n\
         }',

      simpleShadowRayTracerOneLight:
         'vec3 traceRays(Ray eyeRay) {\n\
            vec3 accColor;\n\
            Ray ray = eyeRay;\n\
            float meshID;\n\
            vec3 hit;\n\
            vec3 hitNormal;\n\
```

```
              float t = intersection(ray, meshID, hit, hitNormal);
  // TODO: pass each mesh\'s ID and check if the intersection
returns the same one - if so, continue, otherwise, don\'t add color
\n\
              \n\
              if(inRange(t)) {\n\
                 /* Cast shadow ray, if the energy is over the
threshold */\n\
                 int h = ray.hit+1;\n\
                 \n\
                 Material material = getMaterial(meshID);\n\
                 \n\
                 //float leaving = ceil(dot(ray.dir, hitNormal));\n\
                 //material.iOR =
MEDIUM_IOR*leaving+material.iOR*(1.0-leaving);\n\
                 \n\
                 // Shadow\n\
                 vec3 shLight = material.opacity*ray.light*\n\
                      (material.ambient+material.diffuse+\n\
                      material.specular);\n\
                 \n\
                 if(sum(shLight) > 0.0) {\n\
                    /* Get the hit color */\n\
                    vec3 ambient =
ambientLightColor*material.ambient;\n\
                    \n\
                    vec3 lightColor = pointLightColor[0];\n\
                    \n\
                    vec3 toLight = pointLightPosition[0]-hit;\n\
                    float dist = length(toLight);\n\
                    \n\
                    toLight /= dist;\n\
                    \n\
                    vec3 fromLight = -toLight;\n\
                    \n\
                    /* If falloff distance, determine attenuation\n\

  http://imdoingitwrong.wordpress.com/2011/02/10/improved-light-
attenuation/ */\n\
                    float attenuation = 1.0,\n\
                       lightRange = pointLightDistance[0];\n\
                    \n\
                    if(lightRange > 0.0) {\n\
                       float distFactor = dist/\n\
                            (1.0-pow(dist/lightRange, 2.0));\n\
                       \n\
                       attenuation = 1.0/pow(distFactor+1.0, 2.0);\n\
                    }\n\
                    \n\
                    float lightAngle = spotLightAngle[0],\n\
                       lightCosAngle = spotLightCosAngle[0];\n\
```

```
                    \n\
                    /* If spot light, determine if within spot light
cone */\n\
                    if(attenuation > 0.0 && 0.0 <= lightAngle &&
lightAngle <= 180.0) { /* In the range [0-180], exclusive - should it
be [0, 1]->[90, 0]? (see attenuation with spot falloff) */\n\
                       float cosAngle = dot(fromLight,\n\
                          spotLightDirection[0]);\n\
                       \n\
                       /* Which will result in proper attenuation?\n\
                          http://zach.in.tu-
clausthal.de/teaching/cg_literatur/glsl_tutorial/\n\

   http://dl.dropbox.com/u/2022279/OpenGL%20ES%202.0%20Programming%20
Guide.pdf */\n\
                       attenuation *= ceil(cosAngle-
lightCosAngle)*\n\
                          //pow(cosAngle, spotLightFalloff[0]);\n\
                          pow((cosAngle+1.0)*0.5,
spotLightFalloff[0]);\n\
                    }\n\
                    // float cosAngle = dot(fromLight,
spotLightDirection[0]);\n\
                    // \n\
                    // attenuation *= ceil(attenuation)*ceil(180.0-
lightAngle[0])*\n\
                    //    ceil(cosAngle-lightCosAngle)*pow(cosAngle,
spotLightFalloff[0]);\n\
                    //       //pow((cosAngle+1.0)*0.5,
spotLightFalloff[0]);\n\
                    \n\
                    if(attenuation > 0.0) {   /* Don\'t bother
calculating color or casting if outside of light range */\n\
                       /* Compute the phong shading at the hit\n\
                          Ambient is only accumulated once, so do\n\
                          that outside this loop */\n\
                       vec3 rayColor;\n\
                       \n\
                       vec3 diffuse = material.diffuse*lightColor*\n\
                            max(dot(hitNormal, toLight), 0.0);\n\
                       \n\
                       /* TODO: texturing *//*\n\
                       if(material.textureID >= 0) {\n\
                          vec4 texel =
texture2D(textures[material.textureID],\n\
                             getTextureCoord(meshID, hit));\n\
                          \n\
                          diffuse *= texel.rgb*texel.a;\n\
                       }*/\n\
                       \n\
                       rayColor += diffuse;\n\
```

```
                        \n\
                        vec3 reflection = reflect(toLight,
hitNormal);\n\
                       float highlight = max(dot(reflection, ray.dir),
0.0);\n\
                        vec3 specular =
material.specular*lightColor*\n\
                             pow(highlight, material.shine);\n\
                        \n\
                        rayColor += specular;\n\
                        rayColor *= material.opacity;\n\
                        \n\
                        if(sum(rayColor) > 0.0) { /* Don\'t bother
casting if it\'s just black anyway  */\n\
                           /* TODO: take jittered samples for soft
shadows? */\n\
                           ray = Ray(hit, toLight, h,\n\
                              shLight*rayColor*attenuation,\n\
                              material.iOR, dist);\n\
                           \n\
                           t = intersection(ray, meshID, hit,
hitNormal);\n\
                           \n\
                           /* Calculate color from closest
intersection */\n\
                           if(ray.tLight <= t || t < EPSILON) { //
Light closer or no hit\n\
                              accColor += ray.light;\n\
                           }\n\
                           \n\
                           // accColor += ray.light*(ceil(t-
ray.tLight)+ceil(EPSILON-t));\n\
                        }\n\
                     }\n\
                     \n\
                     accColor += shLight*ambient*material.opacity;\n\
                  }\n\
               }\n\
               \n\
               return accColor;\n\
            }',

         fragmentMain:
            'void main() {\n\
               gl_FragColor = vec4(traceRays(Ray(cameraPosition,\n\
                     normalize(pos-cameraPosition),\n\
                     0, vec3(1.0), MEDIUM_IOR, -1.0)), 1.0);\n\n'+
               THREE.ShaderChunk["fog_fragment"]+'\n\n'+
               /* TODO: alter traceRays to write the meshID of the
                  first intersection into a variable, which can then
                  be assigned to the alpha part of the fragment color
```

```
                This could then be read back on the CPU
                Finally, the FBO texture could be rendered out to
                the screen with all alpha set to 1.0 in a post pass
                void readPixels(GLint x, GLint y, GLsizei width,
GLsizei height, GLenum format, GLenum type, ArrayBufferView? pixels)
*/
            '}'
     });
```