1/7/2022

Assessment Task 3: Major Project Documentation

NUKO

Contents

0.0 Preface

This is a reupload of my supporting documentation for my Major Project. Because it was a school project, some areas were just for the purposes of meeting assessment criteria. Because of this, some sections may seem displaced or unnecessary.

1.0 System Evaluation

1.1 System Architecture

Because the goal of this project is ultimately a learning experience, many decisions in design may be influenced by what would be more interesting to approach rather than the most practical choices.

The goal of this project was to create a 3D multiplayer game with the minimal use of any libraries while still creating a fun, playable and mostly unbroken experience.

1.1.1 Framework

The final program has been written in C as opposed to what was originally planned, purely javascript. Specifically, the program uses Emscripten [1] to compile C into WebAssembly (WASM) [2] which can then be run on a browser. The server uses a similar mechanism, having the server, written in C, compiled into WASM which is then run by NodeJS [3].

To create a window, the program uses SDL [4], a cross platform library to interface with keyboard, audio and OpenGL. In this case, Emscripten integrates it to create a graphical output on the browser.

Ultimately, this means that the program can be compiled natively for testing but released on the browser such that it can be accessed by many devices.

This change was chosen as Javascript is very loosely typed and as a result, as the program grew, it became more difficult to maintain. C on the other hand offers a very minimal set of basic features which makes it easier to approach a problem while making clear within the code itself, what is going on. Furthermore, by compiling it with emscripten, it is able to retain the advantages of javascript, an insurance of cross platform compatibility as long as the browser complies with the latest standards, while being as clean to develop as C.

1.1.2 Game Logic

For a 3D FPS game to function, several systems such as motion, movement and collision have to work smoothly together to provide an engaging experience.

To simulate a player's motion, the euler method was used. The euler method simply involves adding an object's acceleration to its velocity before adding its velocity to its position. This is a simple but effective method that is able to integrate basic motion accurately while satisfying the constraints of the game, as the game will not require any sort of complex motion.

However, for collision detection, a more complex algorithm used to detect when the player collided with the map. Because the map is already composed of convex hulls, they can directly be used for collision. While a naive solution could be to store each hull individually then linearly iterate over it all, performing a traditional collision algorithm such as SAT [5]. This would, however, waste a lot of computation on hulls very distant from the player. Instead, the map was parsed into Binary Space Partition Tree (BSP) [6] where each node represented a plane of a convex hull, the children node branching to either space behind the plane or in front of the plane. This is a method similar to Stan Melax's solution [7], however, a capsule collider being used instead of a cylinder. To detect collision, a player's capsule would traverse the BSP to determine if the space it inhabited was solid. Once all collisions were determined, their velocity was clipped and position shifted such that the player's capsule no longer intersected with the map.

As a major influence on my game was Source Engine games such as L4D2 or TF2, the player's movement logic was mostly derived from such games to create similar behaviour. This simply involved accelerating the player in a specific way as described in Flafla2's article [8]. By doing this, a movement system similar to Source Engine games was able to be replicated.

These separate systems were tied together using an Entity Component System architecture as originally designed and was effectively implemented. Components such as *Capsule*, *Motion* or *PlayerMovement* would be declared in a flat array and store the necessary variables which the systems would read or modify without a complicated series of structures.

1.1.3 Networking

Another major feature of the game is multiplayer. This has been achieved through the use of HTML5 WebSockets as originally detailed. An advantage of using this is that it can be hosted on the free Cloud Application Platform, Heroku [9]. The service provides free hosting of web services while also providing HTTPs. This meant that as long as I hosted the server on Heroku, a secure and encrypted network connection was guaranteed as opposed to self hosting and risking data sent over the network being compromised.

However, to achieve a mostly synchronised multiplayer experience over the internet, the issue of lag had to be overcome. This could be solved by implementing client prediction and lag compensation as described by Valve's Source Multiplayer Networking article [10].

Because I wanted the game to be server authoritative, meaning that the clients could only agree with the server's state of the game, the client game would have to wait for the server's state of the game. Over an internet connection, this could be over 250 ms, causing a game to feel very unresponsive. The solution implemented to solve this was to "predict" the state of the server ahead of time. By storing the inputs yet to be processed by the server, the client can simulate those actions ahead of time to make the player's action feel immediate. However, the game state may still not be in sync due to other factors such as packet loss. However, deviations are corrected once the client receives the true game state through *server reconciliation*.

Server reconciliation is the process of copying the game state sent by the server into the client's local game state. Future client predictions are then made based on the newly received game state. This corrects all future client predictions to ensure the client's game state does not deviate too much from the server.

However, variance in multiple clients' latency may mean that not all clients are viewing the same state of the game, for example, one player may only be 50 ms behind while another could be 200 ms behind. For a player's action to be fair against other players, in actions such as attacks, their latency will have to be compensated for.

This was solved by implementing Valve's described *lag compensation*. By storing an ID of which state of the game an input was made in within the input sent to the server, by caching previous states of the game, the server can effectively "rewind time" (by looking for a paste game state with the corresponding game state ID sent in the input) and processing the attack in that state of the game.

With this, a smooth and secure multiplayer 3D FPS experience was able to be effectively developed.

1.1.4 Rendering

To generate real time 3D graphics, hardware acceleration was used through OpenGL and WebGL, a browser implementation of OpenGL.

OpenGL divides the process of rendering into geometry and composition. Using the basis that any mesh, shape or volume can be graphically represented using some series of triangles, OpenGL allows for a program to pass a set of triangles to the GPU before drawing them.

Composition is done through shaders, a miniature program passed to the GPU to calculate the position of pixels on a screen in parallelization. By writing shaders, complex effects such as lighting and refraction can be rendered over flat coloured triangles.

Using OpenGL, much of the rendering code was parsing models into vertices which could be passed to OpenGL. Several guides on learnopengl.com [11] detail how to create effects such as Phong lighting, HDR, skyboxes and refraction.

However, the renderer has also been used to describe more complex game phenomena such as bullets or particles. Because simulating several bullets or particles would be a waste of computation, the renderer builds in these effects, mostly through deterministic functions of time describing a transformation.

1.1.5 Utility Modules

Smaller functionalities unrelated to Game Logic, Rendering or Networking have also been developed and used. Most notably, a distinct logging function which takes in more detail than the traditional C printf.

A command system has also been developed. The command system is able to bind functions to certain aliases and parse an input text as a set of commands. This allows for the player to interface with the game without using a GUI.

1.2 System Limitations

1.2.1 Framework

While C is much cleaner to develop and maintain, as a result of its simplicity, it lacks many built in features such as easy string handling or automatic memory management. This has made development in some aspects much slower. For example, parsing text can be dangerous as C does not check for array boundary checks. As a result, when copying strings, one has to be careful as to not leave the program vulnerable to buffer overflows where too large of a string is being written to a smaller array, causing data to be written where it should not be and unexpected behaviour.

1.2.2 Game Logic

Because crucial but complex aspects of game logic such as physics were written by myself, several aspects of the code may still contain bugs, especially in collision. In testing, many bugs and cases of unexpected behaviour may be attributed to collision.

Furthermore, the implementation of ECS could also be argued to be excessive and overly complex. While a wide range of entities containing varying functionality, as was envisioned in the original design, may benefit from an ECS architecture, the final game only really has one type of entity, the player. I could imagine this mostly being contained within a single structure which would lower the complexity. The use of the ECS system has also led to solutions being implemented in a convoluted manner. For example, a more basic implementation

of physics, while meeting the same constraints, could be: for each entity, apply acceleration, check for collision, apply collision, contained within a single function call for each entity, the use of ECS has separated these into separate functions. This may lead to very trivial functions being created such as *bg_particle_tick* which contains only a single line of functionality, adding time to a particle emitter's timer.

1.2.3 Networking

By implementing a server authoritative network, all game data sent by the clients will have to be routed to, processed, and sent back by a single centralised server. This may lower the performance of the server and lead to much higher ping as a result of high bandwidth use. To compensate for this, the framerate of the server had to be lowered to send less game state updates per frame. A consequence of this is that physics simulations become less accurate as the time between each frame increases. This becomes evident as at high velocity, the client is able to phase through walls entirely also as a result of a poor physics implementation.

1.2.4 Rendering

While OpenGL is a very powerful tool for rendering graphics, there exists an issue of compatibility. This was evident when the computers at school did not easily support the use of OpenGL in the game and resorted to CPU rendering which made the game run at a very slow pace.

Another limitation of the rendering engine is its over reliance on rendering phenomena such as the motion of bullets, being built into the renderer itself. This invites future problems for scalability, when, potentially, more complex bullet patterns may want to be implemented.

2.0 User Evaluation

2.1 User Feedback Methods

Throughout the stages of testing, various methods of user evaluations were used to gain user feedback.

An early crude method was used throughout the early stages of development. Because this was a multiplayer game, multiple people would need to play the game at the same time to truly test if it was working authentically. Gathering active participants was done by messaging friends on social media for a quick test. This was effective and provided an immediate way to test if features, especially in a multiplayer game, worked consistently on multiple devices.

Figure 2.1.1 - Request for test from developer

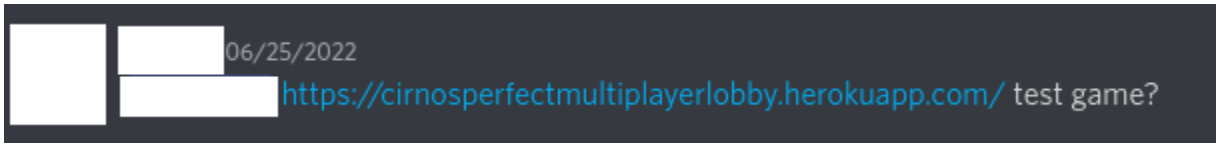
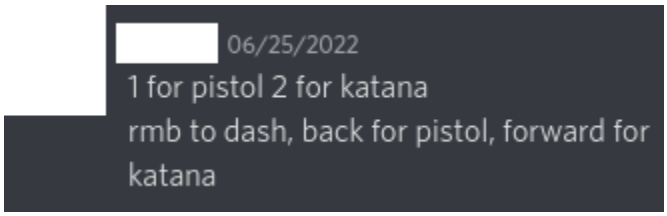
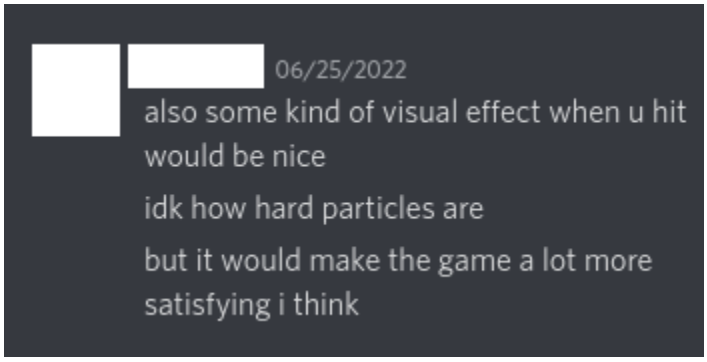


Figure 2.1.2 - Explanation of prototype from developer



2.1.3 Tester - Live feedback given by tester



This was an exchange between myself and a willing participant to test a new feature for an unfinished prototype. The more direct approach allowed for a more open dialogue where I could respond to queries posed to me immediately. I also found that more direct forms of testing provided the live reaction of the tester, and a more genuine reaction.

However, a disadvantage of this type of User Evaluation is that, because the game being tested is still a prototype, it does not have an in-game help system yet. As a result, shown in **2.1.2**, the developer often has to explain to the tester what is going on. This may lead to future biases if they were to test future prototypes as they may already have an idea of what the game is like, distorting an evaluation of what the experience of the game may be like for new players.

However, as the game was being finished, a more formal method of user evaluation was used, the use of a google feedback form. This was given to several play testers, both new and old, to evaluate how effective the in-built help systems were for completely new players and to modify as necessary.

2.2 User Feedback

Google Form: [REDACTED]

Google Form Results:

[REDACTED]

2.3 User Feedback Evaluation

Currently, the user feedback seems to show that the game that was created has met the needs of the user. It has created an engaging 3D first person multiplayer shooter. When asked if testers knew what was happening in the game, one tester succinctly sums up the intent originally outlined, a "free for all pvp game". When asked if the mechanics were understood, most testers responded with yes while the help-guide provided minimal detail on how to play FPS games. This shows that the testers are able to carry over conventions from other FPS games and have projected it onto the game.

This may, however, be a biassed perspective as the people surveyed were mostly later teenagers, the target demographics of FPS games. While this further affirms the notion that the game has successfully created an FPS game, the survey may not properly answer questions such as if the controls were truly intuitive or not.

Furthermore, by examining the user feedback from the google forms, it is evident that lag is still an issue despite solutions to attempt to solve it. However, in the more spontaneous test sessions outside of google forms reviews, I found lag not to be too major of an issue. Especially because of its difficulty to fix (given factors outside of my control) I did not find it warrant a deep investigation into.

On the other hand, the reception to the game seems to be more positive, with players wanting to see more content from the game. Furthermore, from live test sessions, it was found that the players would generally engage with the game for quite a while. However, soon, the game would have cycled through all its content leaving the players disengaged.

Therefore, from user feedback, the direction to take the game seems to be to add more content, perhaps a more engaging system such as persistent ranks or some account system such that the player can gain a more permanent reward. Either way, more engaging content should be developed.

3.0 Source Code

Because the project is quite large, spanning several files, only snippets of code will be shown and explained in the following section.

The entire source code can be seen on github: https://github.com/vv4t/nuko

Several references will be made to the folder structure seen in the github repository.

3.1 Project Structure

/assets contains the assets used by the game.

The main code for the game, written in C, can be found in the **/code** folder. This folder is further broken into smaller folders categorising the module it belongs to.

/tools contains tools written, not necessarily in C, to generate pre-compiled files for the game. Currently, it contains NodeJS scripts which can convert **.obj** [12], a 3D file format, into custom the **.map** and **.mdl** asset files loaded in by the game.

As described in the software architecture section, the game also compiles to a browser compatible format. **/web** contains HTML, CSS and JS used as templates by Emscripten to generate its final output from the C code.

The compilation of these folders are automated using Makefiles which combine several processes required for compilation into a single command.

3.2 Features of C code and styling

struct - Denotes a collection of variables - a class without functions

{type} {alias}([parameters]) { ... } - A function

{type} {alias}[{expression}] - Array declaration

typedef - A shortcut to defining a larger type, e.g. **struct node** -> **typedef struct node node_t**

{alias}_t - typedef'ed type i.e. struct or enum, usually a struct

{function}_R - A recursive function

{variable}.{child} - accessing child member of a non-pointer struct

{variable}->{child} - accessing child member of a pointer to a struct

float - 32 bit floating decimal

char - 8 bit integer, usually used to denote a character such as 'A'

pointer - points to a certain location in memory, generally somewhere in an array, a block of memory, or reference to a variable.

char * - a char pointer is generally a string

const * - the contents cannot be changed through this pointer

if ((...entities[i] & SYSTEM_NAME) != SYSTEM_NAME) - use of bitmask operators to check if an entity state matches the components used by the system

void - when used as a function return, it denotes that nothing is to be returned. When used as a pointer data type, denotes a pointer to any type of data

3.3 Project Code Snippets

The location of the snippet will be labelled using the following format:
::[/dir/to/file.c]:[line]:[function if applicable]

Figure 3.3.1 - The main game logic per-frame systems shared by client and server

```
void bg_update(bgame_t *bg)
{
  // Run all systems
  // NOTE: order is important
  bg_pm_attack(bg);
  bg_pm_free_look(bg); // camera rotation i.e. look
  bg_pm_walk_move(bg); // walk
  bg_motion_gravity(bg); // apply gravity
  bg_motion_integrate(bg); // integrate motion
  bg_clip_capsule_bsp(bg); // detect capsule collision with map
  bg_motion_clip(bg); // clip velocity for collided capsules
  bg_pm_check_pos(bg); // check the current state of player movement
  bg_pm_check_weapon(bg); // check which weapon the client has equipped
  bg_particle_tick(bg); // update particle emitters
}
```

Figure 3.3.2 - Linear search for empty edict entry

To allocate a new entity, the function first has to find an empty slot. This is done through a linear search where an entity state of 0 denotes that it is empty and free.

```
::/code/game/edict.c:15:edict_add_entity()
for (int i = 0; i < edict->num_entities; i++) {
  if (!edict->entities[i]) { // This is simply if the entity state is '0' i.e. no components
    edict->entities[i] = state;
    return i;
  }
}
```

Figure 3.3.3 - Binary tree traversal to detect if a capsule is colliding with the map

A binary tree traversal using a recursive function. This function checks which planes a capsule is colliding with and walks the child node. If the capsule is, at all, in front of the plane, traverse the child of nodes in front of the plane, vice versa. The result is a set of planes colliding with the capsule written into **clip->planes[]**

```
::/code/game/bsp.h:21
// A BSP node
```

```
typedef struct bsp_node_s {
  bsp_brush_t       type;
  plane_t           plane;

  // Rather than left or right in a traditional tree, a plane can be divided
  // into the space behind it or in front of it
  struct bsp_node_s *behind;
  struct bsp_node_s *ahead;
} bsp_node_t;


::/code/game/bg_clip.c:3
void clip_capsule_bsp_R(
  bg_clip_t             *clip,
  const bg_capsule_t    *capsule,
  const bg_transform_t  *transform,
  const bsp_node_t      *node,
  const plane_t         *min_plane,
  float                 min_dist)
{
  if (!node)
    return;

  float top_plane_dist = vec3_dot(transform->position, node->plane.normal) -
node->plane.distance;
  float bottom_plane_dist = top_plane_dist - capsule->height * node->plane.normal.y;

  float min_plane_dist = fmin(top_plane_dist, bottom_plane_dist) - capsule->radius;
  float max_plane_dist = fmax(top_plane_dist, bottom_plane_dist) + capsule->radius;

  if (max_plane_dist > 0.0f) // If the capsule is above the plane
    clip_capsule_bsp_R(clip, capsule, transform, node->ahead, min_plane, min_dist); // Test
the nodes "in front" of it

  if (min_plane_dist < 0.0f) { // If the capsule is below the plane, it is colliding
    // Find the plane where the capsule is colliding the least
    if (min_plane_dist > min_dist) {
      min_plane = &node->plane;
      min_dist = min_plane_dist;
    }

    // If the node/brush is solid, mark the collision
    if (node->type == BSP_BRUSH_SOLID) {
      clip->planes[clip->num_planes] = *min_plane;
      clip->num_planes++;
    }

    // Continue traversing nodes "behind" it
    clip_capsule_bsp_R(clip, capsule, transform, node->behind, min_plane, min_dist);
  }
}
```

Figure 3.3.4 - An insertion sort to display the player's scores

Use an insertion sort the scores of the players. The advantages of this sort are discussed in the comments.

```
::/code/server/sv_game.c:163:sv_print_score()
// Use an insertion sort to sort the score
// This is advantageous as we do not "know" all the elements in the array
// yet. Instead we insert elements as we sort the elements.

// The first loop iterates over all entities
for (int i = 0; i < sv.edict.num_entities; i++) {
  // As seen here, not all elements discovered in the first loop may need a
  // scoreboard entry
  if ((sv.edict.entities[i] & SV_PRINT_SCORE) != SV_PRINT_SCORE)
    continue;

  ent_score[num_ent_score++] = i;

  // The second loop iterates over all the discovered scoreboard entities
  // It is assumed to already to be sorted in descending order from left to
  // right
  for (int j = num_ent_score - 2; j >= 0; j--) { // Start from the last element and loop
backwards
    // Swap the entry until it is neither higher nor lower its neighbours
    // left to right respectively
    if (sv.score[ent_score[j]].kills < sv.score[ent_score[j + 1]].kills) {
      entity_t tmp = ent_score[j];
      ent_score[j] = ent_score[j + 1];
      ent_score[j + 1] = tmp;
    } else {
      break;
    }
  }
}
```

Figure 3.3.4 - Euler integration

Essentially applies 'position = position + velocity * delta_time' to all entities with motion component

Where velocity has already been accelerated by other systems

```
::/code/game/bg_motion.c:51
#define BG_MOTION_INTEGRATE (BGC_MOTION | BGC_TRANSFORM)
void bg_motion_integrate(bgame_t *bg)
{
  for (int i = 0; i < bg->edict->num_entities; i++) {
    if ((bg->edict->entities[i] & BG_MOTION_INTEGRATE) != BG_MOTION_INTEGRATE)
      continue;
    vec3_t delta_pos = vec3_mulf(bg->motion[i].velocity, BG_TIMESTEP);
    bg->transform[i].position = vec3_add(bg->transform[i].position, delta_pos);
```

```
  }
}
```

Figure 3.3.5 - Gravity

Applies acceleration of gravity to all objects with motion component

```
::/code/game/bg_motion.c:40
#define BG_MOTION_GRAVITY (BGC_MOTION)
void bg_motion_gravity(bgame_t *bg)
{
  for (int i = 0; i < bg->edict->num_entities; i++) {
    if ((bg->edict->entities[i] & BG_MOTION_GRAVITY) != BG_MOTION_GRAVITY)
      continue;

    bg->motion[i].velocity.y -= BG_GRAVITY * BG_TIMESTEP;
  }
}
```

4.0 Testing and Evaluation

4.1 Testing Methods

4.1.1 Debugging Statements

Because C was used, the compiler offers no runtime debugging at all. If the program crashes, it will close without providing an error message. As such, it was necessary to create an in-built log system, as seen in **/code/common/log.h** to detect and print errors.

This section of code ran into an error due to the shader not compiling correctly because of a syntax error. The game then outputs the syntax error caused by the shader compilation before safely exiting the program. NOTE: these are terminal outputs.

```
sanakan@netsphere:~/g/nuko/build/linux$ ./nuko
[ERROR] compile_shader(): failed to compile shader
0:2(24): error: syntax error, unexpected invalid token, expecting ';'

[ERROR] shader_load(): failed to load fragment shader
[ERROR] r_hdr_int(): failed to load shader
[ERROR] r_init(): failed to initialise HDR
[FATAL] cl_init(): failed to initialise renderer
sanakan@netsphere:~/g/nuko/build/linux$
```

The game code that outputted the debugging message

```
::/code/client/gl.c:27:compile_shader()
glGetShaderiv(*shader, GL_COMPILE_STATUS, &success);
if (!success) {
  glGetShaderInfoLog(*shader, 1024, NULL, info);
```

```
  log_printf(LOG_ERROR, "compile_shader(): failed to compile shader\n%s", info);
  return false;
}
```

Another example of debugging statements picking up potential run time errors

```
sanakan@netsphere:~/g/nuko/build/linux$ ./nuko
[ERROR] file_read_all(): failed to read file 'assets/shader/hud.vs'
[ERROR] file_read_all(): failed to read file 'assets/shader/hdr.fs'
[ERROR] compile_shader(): failed to compile shader

[ERROR] shader_load(): failed to load vertex shader
[ERROR] r_hdr_int(): failed to load shader
[ERROR] r_init(): failed to initialise HDR
[FATAL] cl_init(): failed to initialise renderer
sanakan@netsphere:~/g/nuko/build/linux$
```

Furthermore, as shown here, sections where errors may occur will recursively return error messages, listing the function where the error occurred. This effectively creates a stack trace where I can trace the functions leading up until the error.

Logging statements can also be used outside of errors. For example, in detecting if game logic is working properly

This is a debugging statement produced by the collision detection code check if it was working before a proper collision response had been implemented. The player had spawned inside a cube before phasing out of it. The debugging statements picked it up and showed

```
sanakan@netsphere:~/g/nuko/build/linux$ ./nuko
[DEBUG] bg_clip_capsule_t(): COLLIDING!!
[DEBUG] bg_clip_capsule_t(): COLLIDING!!
[DEBUG] bg_clip_capsule_t(): COLLIDING!!
[DEBUG] bg_clip_capsule_t(): COLLIDING!!
[DEBUG] bg_clip_capsule_t(): NOT COLLIDING!!
[DEBUG] bg_clip_capsule_t(): NOT COLLIDING!!
[DEBUG] bg_clip_capsule_t(): NOT COLLIDING!!
[DEBUG] bg_clip_capsule_t(): NOT COLLIDING!!
sanakan@netsphere:~/g/nuko/build/linux$
```

4.1.2 Breakpoints

By stopping in the middle of my program, I could analyse the content of my variables to attempt to fix errors.

To do this I used the GNU Debugger (gdb) [14].

The error in my program was that the skull did not die from a single katana hit which was what was intended. Because my code base had become quite large at this point, it had become

somewhat difficult to trace where the program would go just through inspection, so I opted to use a breakpoint.

I set up a breakpoint at *weapon_attack_katana()* which was able to stop the game right before it did the necessary calculations to remove health.

```
sanakan@netsphere:~/g/nuko/build/linux$ gdb
(gdb) file nuko <- load the game file
Reading symbols from nuko...
(gdb) break weapon_attack_katana <- set up the break point
Breakpoint 1 at 0xf2c0: file code/game/../common/nk_math.h, line 96.
(gdb) run
Starting program: /home/sanakan/g/nuko/build/linux/nuko
...
(gdb) step <- step through the program
0x00005555555632c6 in vec3_mulf (b=<optimized out>, a=...)
    at code/game/../common/nk_math.h:104
104       return vec3_init(
(gdb) step
33        vec3_t weap_origin = vec3_add(weap_pos, vec3_mulf(weap_dir, 0.25));
(gdb) step
0x00005555555632e0 in vec3_mulf (b=<optimized out>, a=...)
    at code/game/../common/nk_math.h:104
104       return vec3_init(
(gdb) step
33        vec3_t weap_origin = vec3_add(weap_pos, vec3_mulf(weap_dir, 0.25));
...
(gdb) step
137             cg.bg.health[j].now -= weapon_attribs[cg.bg.weapon[i]].damage;
(gdb) print cg.bg.health[j].now
$2 = 90 <- The weapon damage was that of the pistols?
(gdb) print cg.bg.weapon[i]
$1 = BG_WEAPON_PISTOL <- For whatever reason, even with the katana visually equipped on the
client, the server thought the player had a pistol equipped
```

While stepping I occasionally printed the variables in the debugger to check if it was what I was still expecting. When it reached the line *cg.bg.health[j].now -= weapon_attribs[cg.bg.weapon[i]].damage,* I printed the attacked player's health to realise that it the katana attack had done the same weapon as a pistol. To confirm this, I printed which weapon the player was supposedly using and it was the pistol despite the fact that on my client-side, I had the katana equipped. With this, I was able to locate the bug more accurately and fix it.

4.1.3 Test Units/Expected Results

Though I did not have the hindsight to design my code with testing units in mind, using test units to confirm that modules still output the expected results is an effective method of maintaining a project. As the project gets larger, relationships between modules may also become more ambiguous where a small change in one module may have unforeseen consequences.

Test units can be made by setting up code to run certain sections of code using stub data then compare it with a pre-written set of expected results.

Combined with using Makefile to automate project maintenance, test units could become an effective and efficient way to quickly test small changes across the entire program.

4.2 Method effectiveness

Though breakpoints can be useful when the sequence of code is not entirely predictable, it may be advantageous to use breakpoints and slowly step through the program.

However, debugging statements are far more simple to add. And in most cases, debugging statements were enough to point me towards the source of the error, even if not logged.

Furthermore, the use of gdb requires knowledge of the program which I had not been too well versed in. Thus, I found overall debugging statements to be more effective.

5.0 User Documentation

5.1 Requirements

Processor: 2.0Hz Dual Core or better

GPU: Dedicated GPU with OpenGL 3.0 ES support

Memory: 1024 MB RAM Available

Browser: ES6/WASM/WebGL2 Compatible

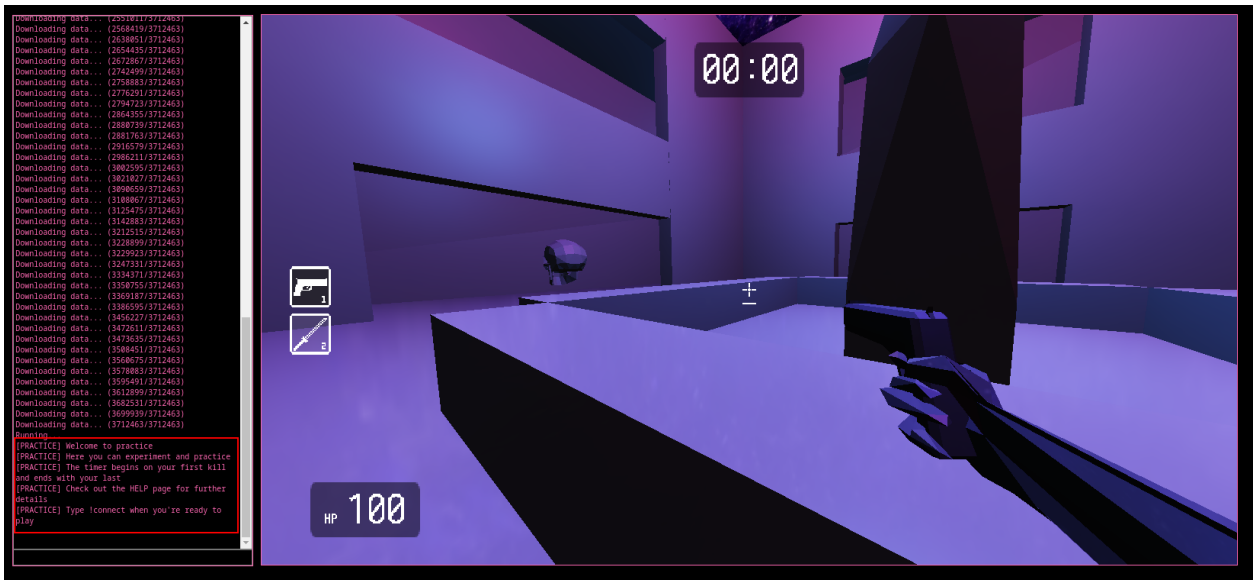
5.2 Installation Guide

The game requires no installation. The official version of the game can be played by simply visiting the following website: https://cirnosperfectmultiplayerlobby.herokuapp.com.

5.3 Gameplay Guide

When you first open the game by loading the website, you will be spawned into the singleplayer practice level.

Here you can practice the basic gameplay mechanics before joining the multiplayer lobby.



The left hand side shows the chat which is an important part of the game as much of the information is displayed through here. The chat also acts as a command system for the player which provides useful features to the player.

The basic controls are:

W | **A** | **S** | **D** - forward | left | back | right

1 | **2** - Pistol | Katana

SPACE - jump

LMB - attack

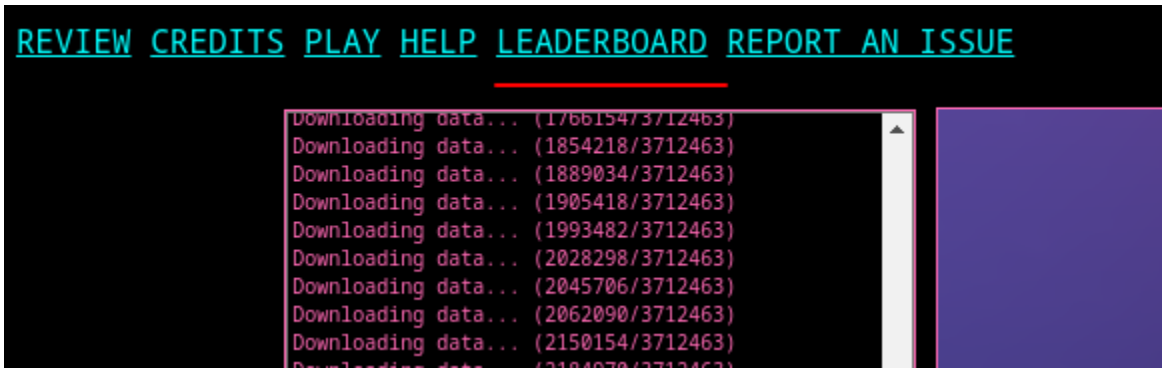
RMB - dash/dodge

Y - chat

In the practice level, attempt to kill all of the skulls as fast as you can. The timer begins when you kill your first skull. After you've killed all the skulls, the timer resets and your time is outputted in chat.

```
[PRACTICE] You finished with a time of 14.25s
```

The practice level also has a speedrun leaderboard, which you can navigate to by clicking the link on the top left.



This is a manually maintained scoreboard where you can submit video evidence of a run attempt you have made which will show up on the leaderboard after being reviewed by a moderator. You should watch the gameplay of the top runs to gain an idea of how to master the mobility system.

When you are ready to connect to a multiplayer room, you can type the command *!connect* in chat

Note that the game is multiplayer and you will need another player to join you while you play. Because the game is not very popular, the rooms will be empty at most times of the day.

```
[DEBUG] connecting...
[SERVER] 'GUEST114747' joined the game
[DEBUG] connected
```

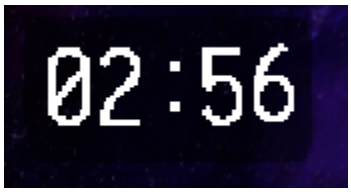
Upon connecting, you will join with the name GUESTXXXXXX. This can be changed using the *!name* command.

```
[SERVER] 'GUEST114747' changed their name to 'bob'
[CHAT] bob: hi
```

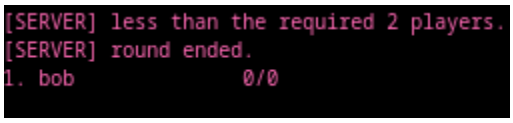
A chat message can be sent by typing any text without the prefix '*!*'.

The game will start when 2 or more players are in the game.

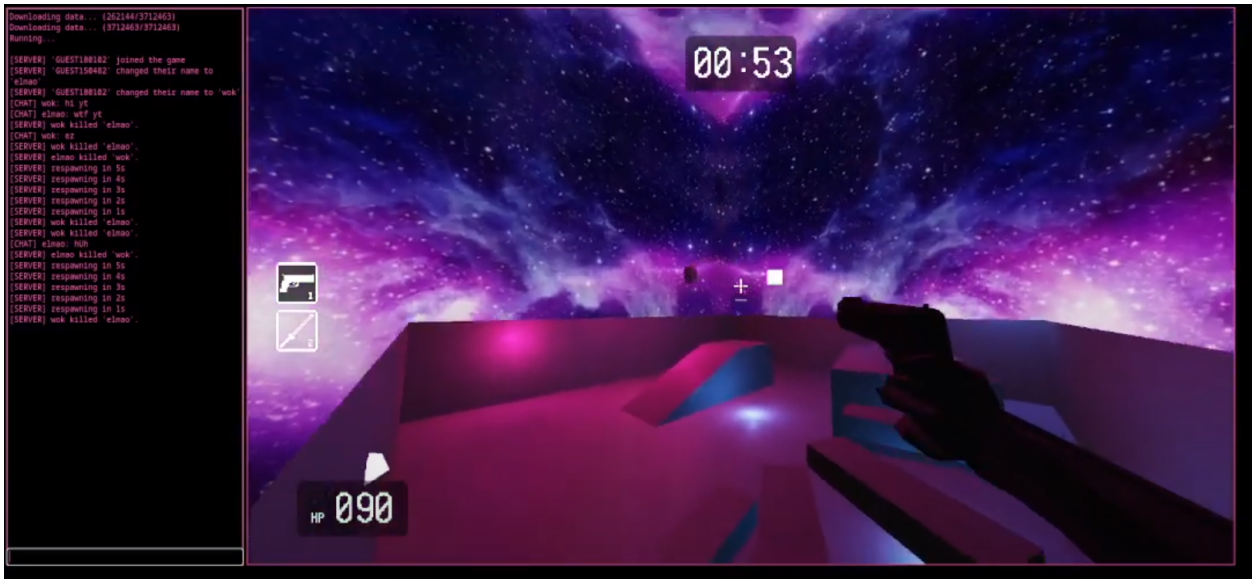
When it starts, the timer at the top will start counting down from 3:00.



However, if this player leaves and there are less than 2 players in the game, the game will automatically stop.



The player now engages in a free for all player vs player battle against each other using the various weapons available. Each kill rewards the player with a point. The round will end when the timer reaches 00:00 and the scoreboard will be outputted in the chat. The player with the most points is the winner and the scoreboard will be displayed in descending order in terms of points



5.4 Advanced Gameplay Guide

A very important mechanic in the game is that of air strafing. Air Strafing allows for a player to control their direction of flight while in the air. This can be done by letting go

of any forward/backward direction key and strafing while looking left and right during flight. This may be better explained in a video teaching this mechanic in another game: https://www.youtube.com/watch?v=mozIx2v4s7s.

Bunny Hopping, known as Bhopping, is also another crucial mechanic in the game. By jumping just as you land, you can preserve your momentum as long as you continue being airborne. When used with air strafing, this allows for the player to become very mobile.

Bunny Hopping and air strafing can further be combined with the katana dash forward in order to maintain controlled movement at very fast speeds. Top runs on the leaderboard demonstrate the effective use of this mechanic.

The weapons can also be better understood in terms of their play style.

The gun applies a small amount of damage from a distance and a dash which propels you backwards. The style of play being emphasised here is that of kiting and running away from pursuers while still damaging them.

The katana applies an extremely large amount of damage at short range, killing any other players instantly even with full health along with a forward dash which allows for the attacker to chase other players. The style of playing being emphasised here is that of engaging and pursuing or sudden attacks by bursting forward with great speed.

While these two have their respectively distinct playstyle, because the player can switch between them immediately, adapting only a single playstyle per round is not recommended. Instead, the player should master switching between both to adapt to the situation.

For example, if a pursuit is becoming futile, the pursuing player can instead switch to the gun to deal ranged damage. From there, the fleeing player is forced into the role of the aggressor, either remaining using the gun where the battle turns into a firefight, or switching to the katana to reverse the roles. Furthermore, the attacking player switching to the gun could also be a tactic to draw the pursuing player in. Though such scenarios may not play out as such in a game, a player should switch between weapons strategically and look for openings where one or the other could be more effective.

This is a video of two relatively experienced players in the game and demonstrates some of the gameplay mechanics and strategies at hand: https://www.youtube.com/watch?v=nSL_Yyi5cRs

5.5 Configuration

As the game currently does not have an in-built GUI system, configuration is done through the command system. Even so, because of time restraints, the only configurable command available is to adjust sensitivity. This can be done through sending the following in chat:

```
'!sensitivity' <- display the current sensitivity
0.005
'!sensitivity 0.001' <- lower the sensitivity accordingly
```

The system is very ineffective and future revisions will be made.

5.6 Local Server Installation

This guide is for installing a local, custom server for the more technically experienced.

NOTE: I did not have a windows machine I can test this on. As such, the following instructions have only been tested on ArchLinux

5.6.1 Precompiled Server Binaries

The precompiled WASM server files can be found on https://github.com/vv4t/nuko/releases

Prerequisites:

- *NodeJS https://nodejs.org/en/download/*

To set it up:

1. Download the latest zip file and unzip it.

2. Open a terminal and navigate to the unzipped folder.

3. Type the following commands: '*npm install* .' and '*node* .'

```
sanakan@netsphere:~/tmp/web$ npm install .

added 58 packages, and audited 59 packages in 2s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
sanakan@netsphere:~/tmp/web$ node .
Listening on ::8000
warning: no blob constructor, cannot create blobs with mimetypes
warning: no BlobBuilder
warning: Browser does not support creating object URLs. Built-in browser image decoding will
not be available.
net_listen(): unimplemented
```

A HTTP server will now be listening on port 8000. To navigate to this site on the browser, simply head to the url http://localhost:8000/.

5.6.2 Compiling from source

The source code is publicly available at the project's github repository at https://github.com/sanakanw/nuko and can be downloaded and modified.

Prerequisites:

- *Git https://git-scm.com/book/en/v2/Getting-Started-Installing-Git*

- *NodeJS https://nodejs.org/en/download/*

- *Emscripten https://emscripten.org/docs/getting_started/downloads.html*

- *Make https://www.youtube.com/watch?v=5TavcolACQY*

NOTE: Though a linux shell is used here, the same should be possible on Git Bash

1. Clone the repository

```
sanakan@netsphere:~/tmp$ git clone https://github.com/sanakanw/nuko.git
Cloning into 'nuko'...
remote: Enumerating objects: 2128, done.
remote: Counting objects: 100% (560/560), done.
remote: Compressing objects: 100% (385/385), done.
remote: Total 2128 (delta 301), reused 406 (delta 171), pack-reused 1568
Receiving objects: 100% (2128/2128), 54.40 MiB | 3.25 MiB/s, done.
Resolving deltas: 100% (912/912), done.
```

2. CD into the directory

```
sanakanw@netsphere:~/tmp$ cd nuko
sanakanw@netsphere:~/tmp/nuko$ ls
README.md assets code linux.mk tools web web.mk
```

3. Make web.mk

```
sanakanw@netsphere:~/tmp/nuko$ make -f web.mk
```

4. CD into the web build directory

```
sanakanw@netsphere:~/tmp/nuko$ cd build/web
```

5. Install the node packages and run the node script

```
sanakan@netsphere:~/tmp/nuko/build/web$ npm install .

added 58 packages, and audited 59 packages in 2s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
sanakan@netsphere:~/tmp/nuko/build/web$ node .
Listening on ::8000
warning: no blob constructor, cannot create blobs with mimetypes
warning: no BlobBuilder
warning: Browser does not support creating object URLs. Built-in browser image decoding will
not be available.
net_listen(): unimplemented
```

With this, any changes to **/code** can be modified and compiled to run a local custom server.

6.0 Logbook

Date	Detail
03/04/2022 Home	Major Rewrite - Planning to port of previously written prototype to C++ because Javascript lacks strong typing which makes project difficult to maintain - Set up compiling to emscripten for browser play - WASM support - File organisation
04/04/2022 Period 1	Organisation of files and additional rendering code - Restructured some folders - Prepared basic OpenGL rendering - Prepared game components for ECS
05/04/2022 Period 1	ECS bitmask implementation
13/04/2022 Home	Port to C - Reimplementation of prototype in C instead of C++ - Capabilities in C++ (classes, templates, inheritance...) overcomplicate ECS implementation, an approach in C would be cleaner, more straightforward and efficient
13/04/2022 Home	- Implemented better brush grouping such that objects of different textures can be rendered in single draw call
14/04/2022 Home	Project formatting - Aligned struct members, functions, arguments - Better naming
14/04/2022 Home	- Error checking on faulty map loads - Memory management - properly free BSP nodes on deletion

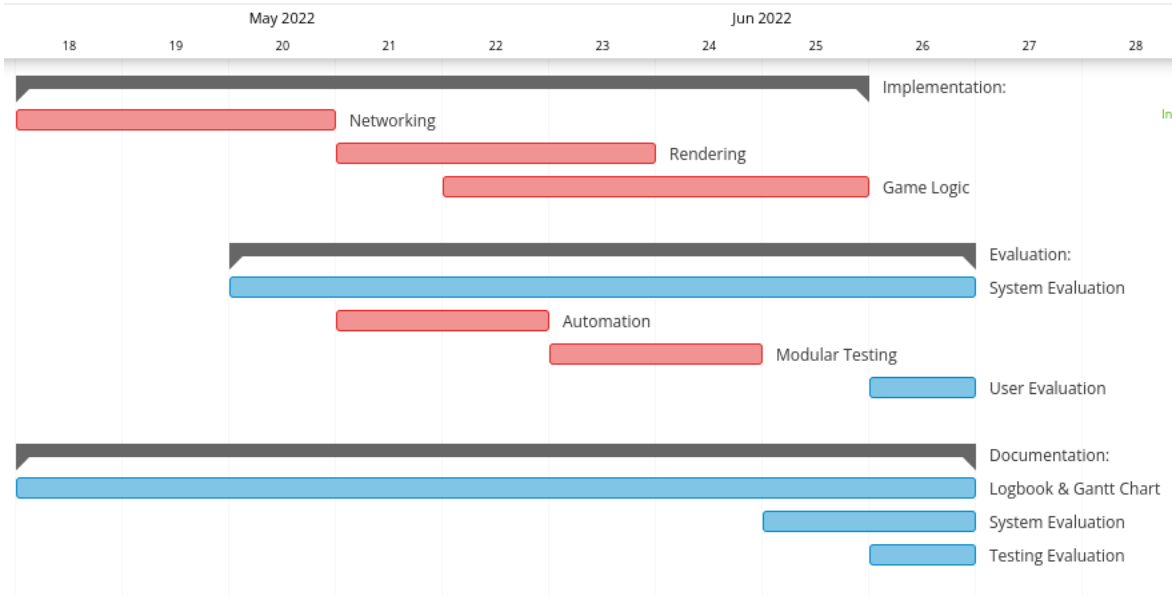
14/04/2022 Home	- Implemented a console system similar to Quake - Commands as aliases for function calls - Able to bind key presses to certain commands - Reimplemented input system such that it's command oriented so that keybindings through the console are possible
15/04/2022 Home	- Reorganisation of code
16/04/2022 Home	Model loading - Created tools to convert obj to a custom .mdl format which is a binary format easier to read in C - Implemented model rendering
17/04/2022 Home	- Further code reorganisation
18/04/2022 Home	Set up networking compatible with emscripten - Client and server - NodeJS server which runs a emscripten compiled WASM binary - Client and server communication through WebSocket achieved through an interfacing JS script - Next will be to implement integrate the networking module into the game - Added a .gitignore to reduce file size of github repo
20/04/2022 Home	- Implemented the networking module into the game - Client is able to send Usercommands to the server while the server responds with a snapshot of the game
24/04/2022 Home	- Added client disconnect detection to remove disconnected clients from being processed
24/04/2022 Home	- Attempted to create a HTTPs websocket using native libraries so that native clients can play on the same server as browser clients, but libraries are difficult to use - In the end, used BSD sockets to create simple TCP connection to server - Able to run a native version of the game - This was done as compiling using emcc was slow and would require compiling the entire project per minor change
25/04/2022 Home	- Automated project building use Makefiles - Compiles each C file individually before linking them together - Removes the need for gcc to compile the entire project in a single go - However, was unable to find a similar method for emcc
29/04/2022 Period 4	- Replaced while(1) loop in the client with fixed update loop to ensure server and client ran at same tickrate - Boundary checks to ensure on server-side that client's usercmd queue was not overflowing - Reorganisation
01/05/2022 Home	Heavy Reorganisation - Cleaned up client code - Static singletons over passing structs everywhere - Fixed update loop - clock was not registering ticks because SDL was putting the program to sleep to wait for the next frame. This was solved by using a MONOTONIC clock

	- Introduction of player shooting
04/05/2022 Home	Heavy Reorganisation - Merged game into client (cl_cgame) and sgame into server (sv_game) - More efficient makefile automation - Lag compensation for shooting by checking which snapshot the usercmd was sent and checking the ray against that snapshot
06/05/2022 Period 2	- Game chat - Cap on camera pitch rotation - Fixed excessive loop update caused by large delta times between frames where the player tabbed in and out which effectively made the delay between subsequent frames 0 until it believed it had processed all the lag
07/05/2022 Home	Console fixes - fixed improper string literal handling - Added '!' to denote console command vs chat command - Moved console input to cl_main using sys_read_in to read from stdin
07/05/2022 Home	- Changed from server in-built commands to individual network packets. Building another command-parser within the server chat would be effectively rewriting the existing command module
08/05/2022 Home	- Added scoreboard - Added respawn 5s - Set lower resolution for chat to fit on page - Created new map - nk_arena
09/05/2022 Period 1	- Researched pre-existing HUD implementations in OpenGL C, however, did not want to overcomplicate project yet - Created a simple HUD system by display flat rectangles from a simple sprite sheet - May need to implement GUI library for more complex GUI arrangements such as chat - Created health system using created HUD system
11/05/2022 Home	- Implementation of Phong lighting - Sensitivity console command - Fixed damage src to properly display kill messages
07/05/2022 Home	Functional game - Added visual bullets - Added round timer - Added timer HUD - Socket disconnect on player leave
13/05/2022 Period 4	More maps - nk_yuu, nk_chito - Host map rotation - Auto disconnect if room is full
14/05/2022 Home	- Moved all frame related functions to game/frame.c - Minimised frame size to data - Comments on client and minor adjustments

23/05/2022 Period 2	- Began code commenting (Have not done any code commenting up until now)
24/05/2022 Period 1	- Code commenting
27/05/2022 Period 4	- Code commenting
06/07/2022 Period 2	- Further commenting
07/07/2022 Period 1	- Further commenting
17/07/2022 Period 2	- More commenting
24/07/2022 Period 4	- Began working on weapon models
25/07/2022 Home	Massive Graphics Update - Fixed weapon models - Added attack2 dashing - Added HDR - Added skybox - Added refraction - Added particles on death
26/07/2022 Home	- Added another map, nk_street
27/07/2022 Period 1	- Repurposed localhost testing into tutorial system - Player can play in a practice map, killing dummy players on a timer before connecting to actual game - Began formal documentation
28/07/2022 Period 2	- Documentation

7.0 Gantt Chart

Gantt Chart of Weeks since January



8.0 References

[1] https://emscripten.org

[2] https://webassembly.org

[3] https://nodejs.dev

[4] https://www.libsdl.org

[5] https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169

[6] https://en.wikipedia.org/wiki/Binary_space_partitioning

[7] https://www.gamedeveloper.com/programming/bsp-collision-detection-as-used-in-mdk2-and-neverwinter-nights

[8] https://adrianb.io/2015/02/14/bunnyhop.html

[9] https://herokuapp.com

[10] https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

[11] https://learnopengl.com

[12] https://en.wikipedia.org/wiki/Wavefront_.obj_file

[13] https://www.gnu.org/s/make/manual/html_node/Introduction.html

[14] https://en.wikipedia.org/wiki/GNU_Debugger