Project:

For this optimisation and porting task I will be using a game I was working on over the summer.
The game is a top down open world multiplayer farming game called Venture.
The game is getting roughly 20-30FPS in debug build, which isn't much considering the content in the game, and in release build the game runs better, but there is memory issues when explore the world.

Engine:

The game was developed in C++ with SDL and used the boost::asio library for the networking. However, for the scope of this project and to ease the port to PS4 I will be removing the networking from the game, and focus on the single player optimisations. Furthermore depending on the support for SDL on the PS4 I may be looking into converting to the PhyreEngine for the rendering as that will also improve the performance.

Game Features:

- Character customisation
- Level saving / loading
- Player saving / loading
- Procedural terrain generation
- A chunk based level system
- Inventory system
- Farming and terrain manipulation

Profiling Tools:

Visual Studio profiler is a very useful tool within visual studio that can profile the CPU, memory and GPU (only using the directX).
RenderDoc may be useful for if I decide to overhaul the SDL rendering and use openGL instead, this will be able to help profile the openGL rendering pipeline.

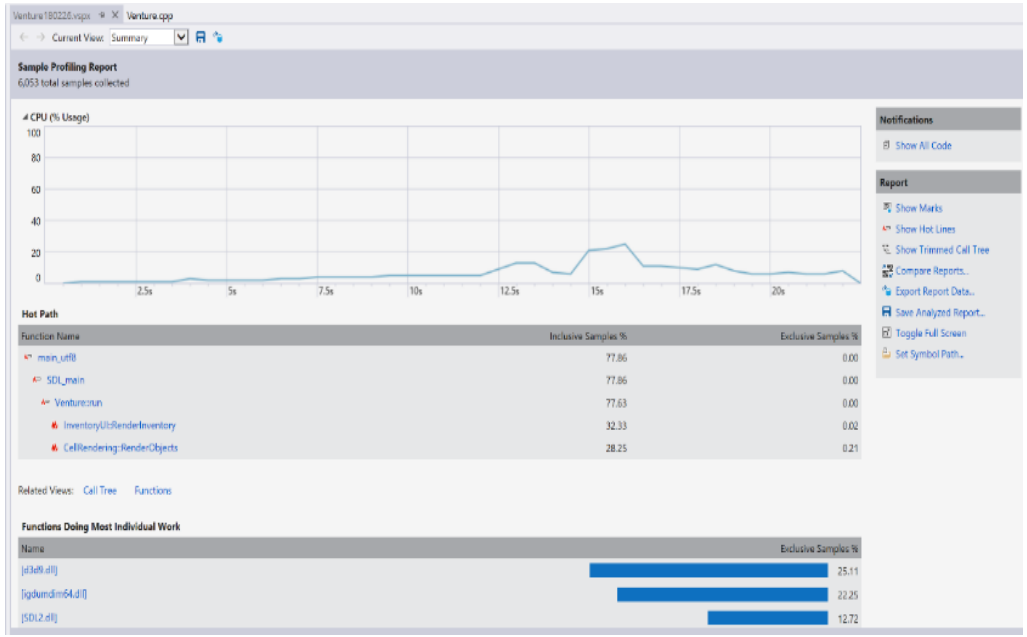
Other Notes:

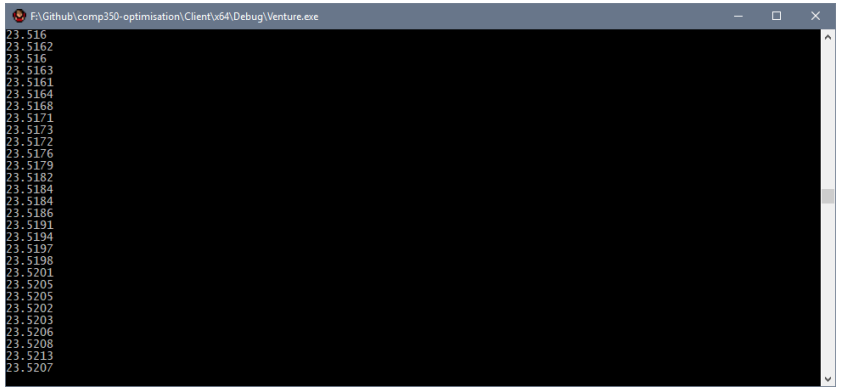
During the optimisation of this game, over the easter period, I spent mostly working on game bugs and fixes, and getting some old code working, which I could optimise. This means that I have structured this document into two sections, one before the new features and fixes, and one after.

Benchmark:

To benchmark I started with the visual studio profiler, this highlighted what functions where using most of the resources.

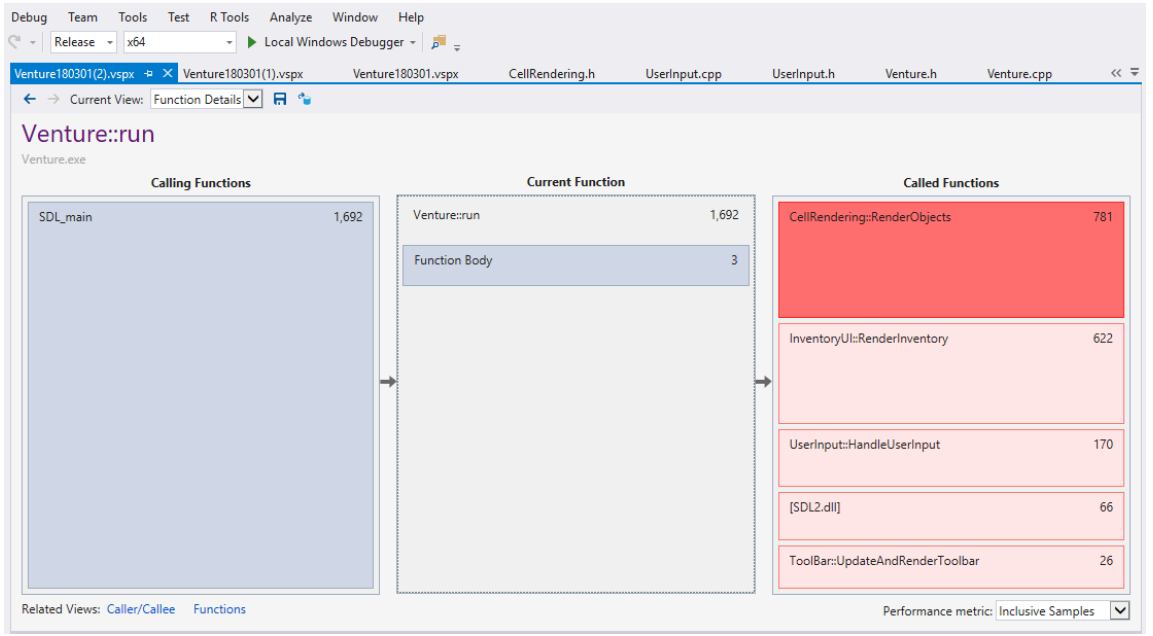
There is a clear spike where I had opened the players inventory, as well as when the level was being generated





Measure:

The key functions that are affecting the performance the most is RenderObjects, which renders the levels cells.
I will start by looking at how the rendering is working, to see why this is an issue.



I also implemented a way to check the time it takes to render a frame, however this only works in debug, where I have access to the console.

<- This is the FPS I am getting when idle in debug.

Detect:

The way the textures get loaded into the game is I created a texture class that loads each texture as a separate image, and it does this for every tile generated.

So each texture loaded in meant that as you explored the world the memory usage will gradually increase and performance would decrease.

This method also meant I had to do a few hacks to get the trees rendering properly. The image to the left shows I had two vectors, one that contained the trees that were to be rendered above the player, and another that was to be rendered below.

This could be avoided by creating a batch renderer.

Moreover, there are a lot of Textures for each game object, which is not maintainable as the game scales.

Solve:

To Solve this issue, I worked on creating a renderAtlas function in the Texture class, this will take an ID for the sprite and get that sprite from the atlas and render it to a point in the game.

To do this, I created a struct that contained the location of the sprite, the atlas index number and the layer.
I also created a vector of all the textures, so that I could batch render all the textures at the end of the frame.

```
private:
    //Target Darkness
    float Tdarkness;
    float darkness = 255;
    float time = 0;
    const std::string RoomSpriteTextureLocation = "Resources\\Sprites\\roomSprites\\texturePack\\";
    const std::string ItemsSpriteTextureLocation = "Resources\\Sprites\\SpawnItems\\";
    const std::string TerrainSpriteTextureLocation = "Resources\\Sprites\\Terrain\\";
    const std::string TreeTerrainSpriteTextureLocation = "Resources\\Sprites\\Terrain\\Trees\\";
    const std::string WallSpriteTextureLocation = "Resources\\Sprites\\Terrain\\Walls\\";

    const std::string characterTextureLocation = "Resources\\Sprites\\Character\\";
    const std::string playerStatsTextureLocation = "Resources\\Sprites\\GUI\\PlayerStats\\";

    // Textures for game objects

    //! Are the textures for the statusbars
    Texture healthBarTexture;
    Texture hungerBarTexture;
    Texture tiredBarTexture;
    Texture oxygenBarTexture;

    //TODO: load json file containing the array of different texture IDs
    int grassID = 5;
    int sandID = 8;
    int waterID = 0;
    int water2ID = 1;

    //! Atlas textures
    Texture terrainAtlas;
    Texture roguelikeAtlas;

    //! Contains the data for texture positions and layers
    struct textureID
    {
        int index;
        int x, y;
        int width, height;

        //! Layer to be rendered on -- 0 = Ground -- 1 = Items on ground -- 2 = Player -- 3 = Above player
        char layer;
    };

    enum layers
    {
        seaLevel,
        ground,
        onGround,
        player,
        abovePlayer
    };
    //! A vector of all textures
    std::vector<textureID> allTextures;
```

I also removed the rendering of the trees issue, by removing the two vectors of trees. Instead I used an enum to store the layer of each sprite, and that decides when the texture will be rendered.

```
void CellRendering::AddToBatchRendering(int ID, int x, int y, int& size, char layer)
{
    textureID texture;
    texture.index = ID;
    texture.x = x;
    texture.y = y;
    texture.width = size;
    texture.height = size;
    texture.layer = layer;
    allTextures.push_back(texture);
}
```

Solve:

The new texture render atlas function

```
void Texture::renderAtlas(SDL_Renderer* renderer, int index, int x, int y
{
    // Create source and destination rects
    SDL_Rect dest;
    SDL_Rect srcrect;
    if (!texture)
    {
        texture = IMG_LoadTexture(renderer, fileName.c_str());

        if (!texture)
        {
            throw InitialisationError("IMG_LoadTexture failed");
        }
    }
    // Atlas Without borders
    if (atlasType == 0)
    {
        // Get the x and y positions based on the amount of tiles in atla
        int sourceX = index;
        int sourceY = 0;
        while (index > atlasTileWidth)
        {
            sourceY += atlasTileSize;
            index -= atlasTileWidth;
        }
        sourceX = index * atlasTileSize;

        // get source rect data
        dest.x = x - width / 2;
        dest.y = y - height / 2;
        dest.w = width;
        dest.h = height;

        srcrect = { sourceX, sourceY, atlasTileSize, atlasTileSize };
    }
    // Atlas With borders
    else if (atlasType == 1)
    {
        // Get the x and y positions based on the amount of tiles in atla
        int sourceX = index;
        int sourceY = 0;
        while (index > atlasTileWidth)
        {
            sourceY += atlasTileSize + 1;
            index -= atlasTileWidth + 1;
        }
        sourceX = index * atlasTileSize + index;

        dest.x = x - width / 2;
        dest.y = y - height / 2;
        dest.w = width;
        dest.h = height;

        srcrect = { sourceX, sourceY, atlasTileSize, atlasTileSize };
    }

    // Copy source texture data into game
    SDL_RenderCopy(renderer, texture, &srcrect, &dest);
}
```

This function gets the location of a sprite from an atlas, based off the index number which is passed in.

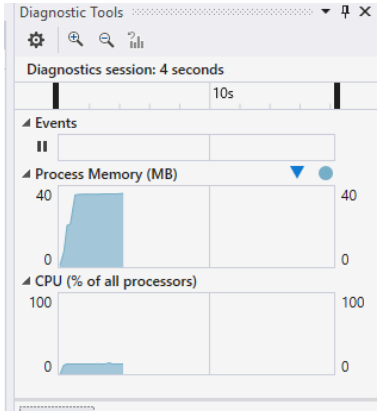
There are two atlas types that I have implemented, one that doesn't have any borders, and one that has a 1 pixel border between sprites.

In the future if there are more than two types of atlas', I will try to streamline this code to be more maintainable, however for the two types that I have this was the easiest way I saw how to implement it.

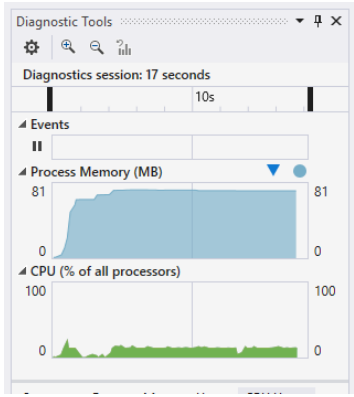
Check:

This change means that now the game doesn't create a new texture for each cell as the player explores the level, which results in a much lower memory footprint.

NEW



OLD



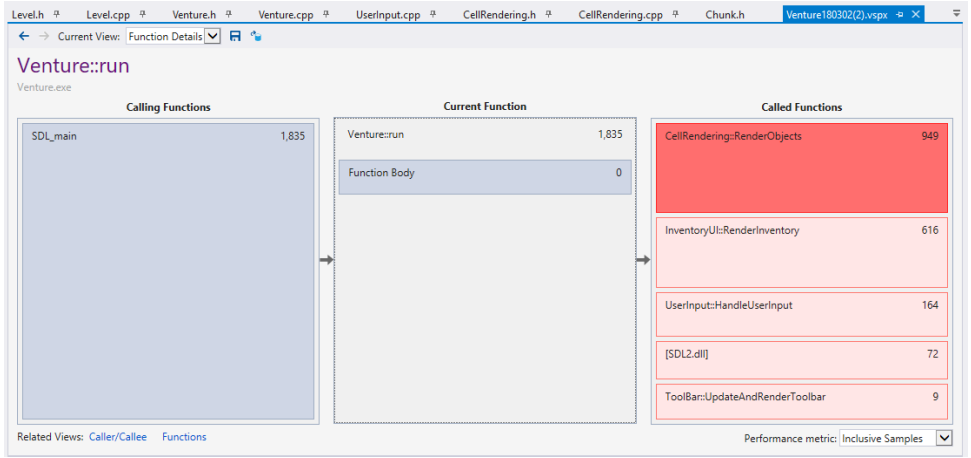
When comparing the results of these changes, this shows that RenderChunk function increased performance by 3.49, which changed from 9.85. I believe the numbers mean CPU processing time, so the new change means that the rendering function used about 6 seconds less CPU processing time.

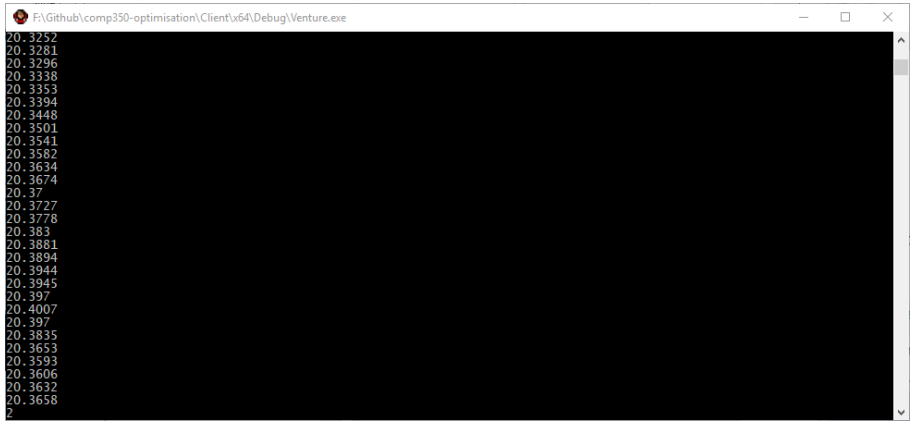
Comparison complete.

Comparison Column	Delta	Baseline Value	Comparison Value
[nvd3dumx.dll]	↑	13.42	30.00	43.42
[win32u.dll]	↑	7.74	0.30	8.04
CellRendering::RenderChunk	↑	3.49	9.85	13.34
Texture::render	↑	1.14	0.00	1.14
[ntdll.dll]	↓	-1.50	6.27	4.77
std::vector<CellRendering::textureID,std::allocator<CellRendering::textureID>...	↓	-2.54	2.54	0.00
[ucrtbase.dll]	↓	-2.63	4.48	1.85
Texture::renderAtlas	↓	-4.63	4.63	0.00
[SDL2.dll]	↓	-4.89	13.58	8.69
[d3d9.dll]	↓	-9.34	19.70	10.36

Benchmark:

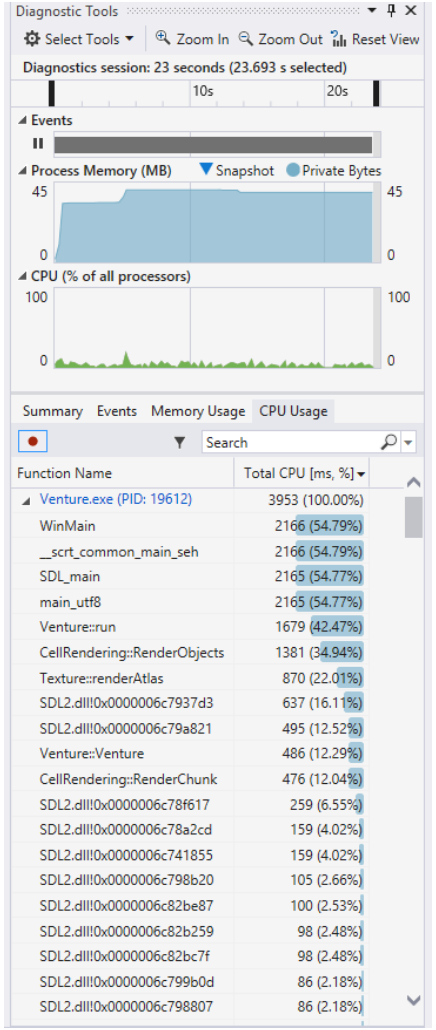
When I ran the test again I still found that render objects is sill using a considerable amount of CPU utilisation.





The frame rate in debug is still not optimal, only getting about 20 FPS.

RenderObjects is still using roughly 35% of the processes overall CPU usage, even though the total CPU usage of the game is very low still, it'll be worth going into depth to try and optimise RenderObjects further, as this is the main cause of the slow performance at the moment.

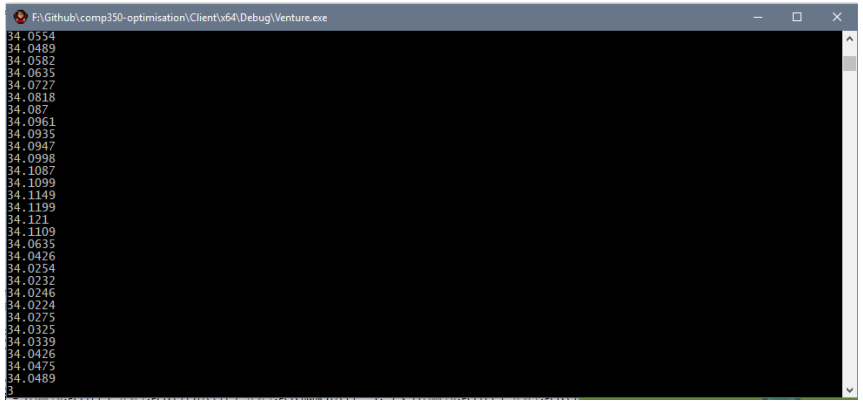


Detect:

The main cause of the lag is this small bit of code below, which renders the chunks that are around the camera position. I remember writing this code very late at night so I will go over it and see if there is any areas that can be optimised.

```
//! Renders the chunks of cells
void CellRendering::RenderObjects(Level& level, SDL_Renderer* renderer, Camera& camera, Player& player, std::vector<std::shared_ptr<Player>>& allPlayers)
{
    // Alter the textures
    AlterTextures(level);

    // Render all the cells in the chunks
    for (int i = (camera.getX() / level.getCellSize()) / level.getChunkSize() - 1; i < ((camera.getX() / level.getCellSize()) / level.getChunkSize()) + camera.ChunksOnScreen.x ; i++)
        for (int j = (camera.getY() / level.getCellSize()) / level.getChunkSize() - 1; j < ((camera.getY() / level.getCellSize()) / level.getChunkSize()) + camera.ChunksOnScreen.y ; j++)
                RenderChunk(level,camera,player, level.World[i][j], renderer);

    // Batch render all the textures
    for each(auto &texture in allTextures)
    {
        roguelikeAtlas.renderAtlas(renderer, texture.index, texture.x, texture.y, texture.width, texture.height );
    }
    allTextures.erase(allTextures.begin(), allTextures.end());
```

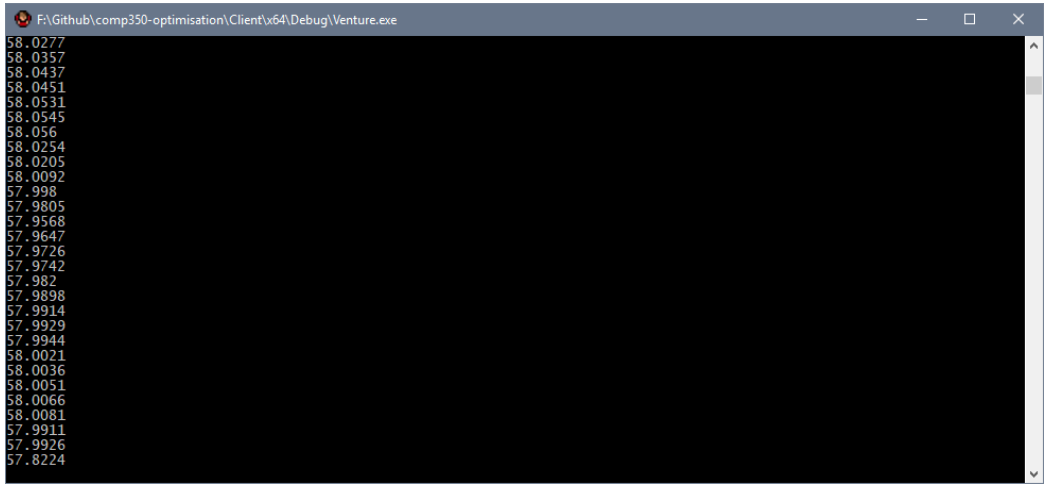
After playing round with the chunk rendering code, I found that it was actually rendering a lot more cells off screen, that I will be able to cull.



All I did was removed 1 from the x and y chunks that were being rendered, and this instantly gave an extra 10+ frames per second.

Solve:

The next thing I did was try to cull as much as possible from outside the players view, to do this I decreased the chunk size, this was extremely surprising.
Previously the chunk size has always been 16, which is 16x16 cells in a chunk. Just to see I reduced the chunk size by half, this gave a huge boost to frame rate.

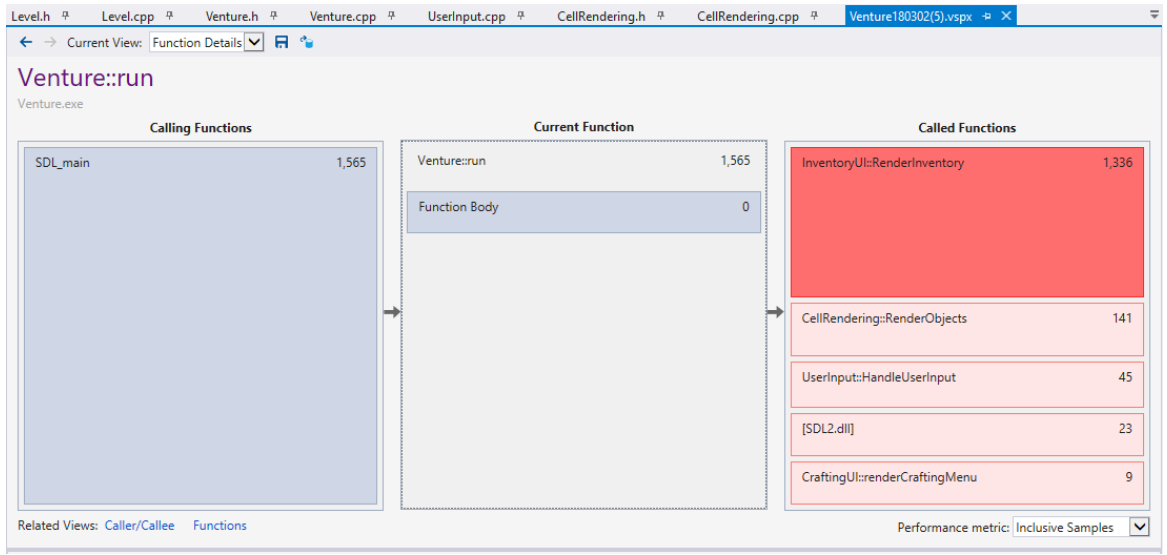


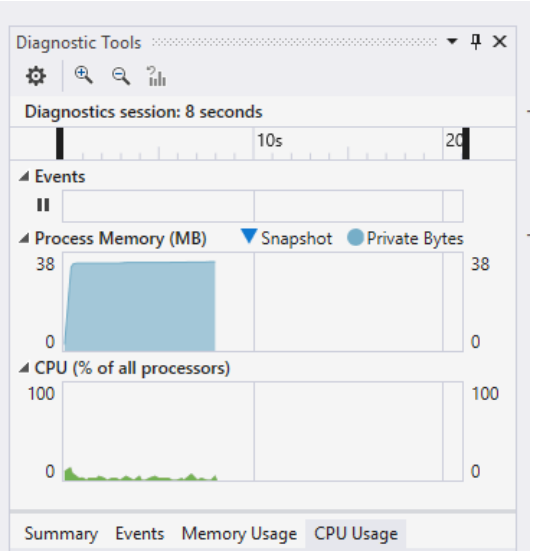
By simply changing the chunk size from 16 to 8 gave an almost 2x change in frame rate!
I tried changing the chunk size by half again, but this actually decreased performance slightly, so the sweet spot seems to be around 8x8 cells per chunk.

I suspect this change may be due to the CPU being able to cache the 8x8 chunks better in the L1 and L2 cache more efficiently.

Check:

Changing the chunk size seems to have made such a difference to performance that, the RenderObjects function is no longer the top called function, and was only called 141 times.





That is quite a drastic change from such a small fix, previously the RenderObjects function was being called roughly 1000+ times, where as it is only being called ~150 times now.

I suspect the reason for why the chunk size affected the performance so drastically, was because with a larger chunk size, there are a lot more cells that will be rendered off screen, which are not being culled, so having smaller chunk sizes means there are less cells that will be rendered offscreen.

Benchmark:

When I was looking testing the frame rate with RenderDoc I found that the frame rate was capped
to 60 FPS, and I figured this was because of Vsync. However the refresh rate on my monitor is 100Hz,
So I started looking at how I could change this.

Detect:

```
// Create the window
window = SDL_CreateWindow("Venture", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, gameSettings.WINDOW_WIDTH, gameSettings.WINDOW_HEIGHT, SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
glContext = SDL_GL_CreateContext(window);
if (window == nullptr)
    throw InitialisationError("SDL_CreateWindow failed");

// Create Renderer
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_PRESENTVSYNC);
if (renderer == nullptr)
    throw InitialisationError("SDL_CreateRenderer failed");
```

I simply changed SDL_RENDERER_PRESENTVSYNC to SDL_RENDERER_ACCELERATED.

Measure:

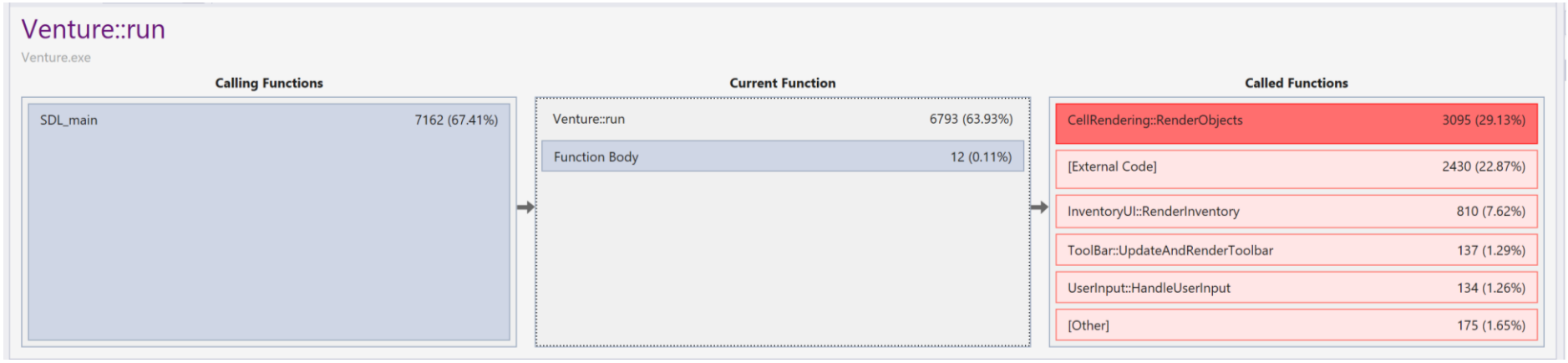
I Tested the four different SDL rendering options, to see which one gave the best frame rate, and from my results SDL_RENDERER_ACCELERATED was the best performance rendering option.

Solve:

```
// Create the window
window = SDL_CreateWindow("Venture", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, gameSettings.WINDOW_WIDTH, gameSettings.WINDOW_HEIGHT, SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
glContext = SDL_GL_CreateContext(window);
if (window == nullptr)
    throw InitialisationError("SDL_CreateWindow failed");

// Create Renderer
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
if (renderer == nullptr)
    throw InitialisationError("SDL_CreateRenderer failed");
```

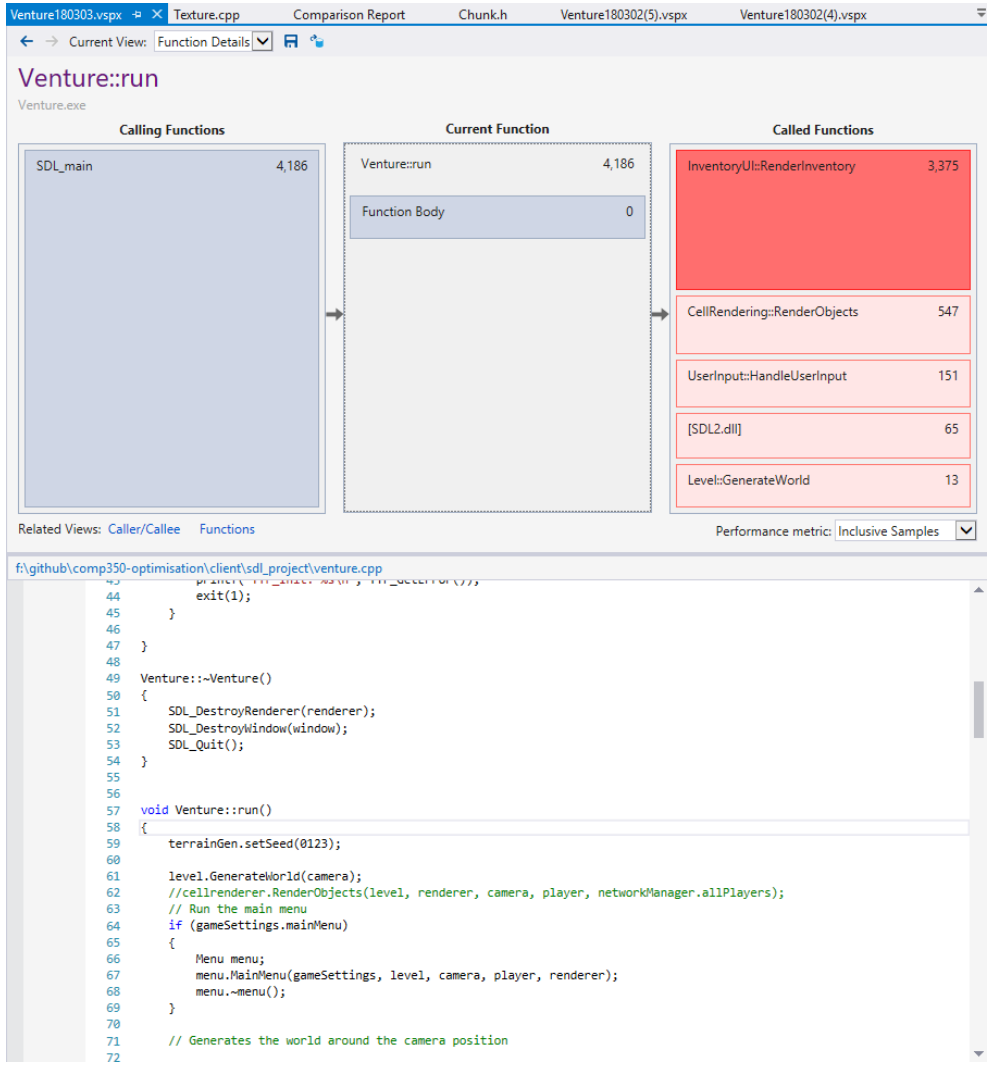
One problem I have with this is that because the frame rate is no longer a steady rate, and starts to max out the system, the game seem to speed up and slow down.
This may cause an issue with screen tearing, if the frame rate goes above the refresh rate of the monitor. I may change this back at some point, if it gets too much of an issue, but for now it is useful for measuring the frame rate when it goes above 60FPS.

Check:



Benchmark:

This is now the current version of the game, and in this game I opened the inventory a few times, and this shows that there is some optimising to be done with the RenderInventory function.



Measure:

RenderDoc now shows I am getting a reliable 60FPS when the game is not being zoomed out.

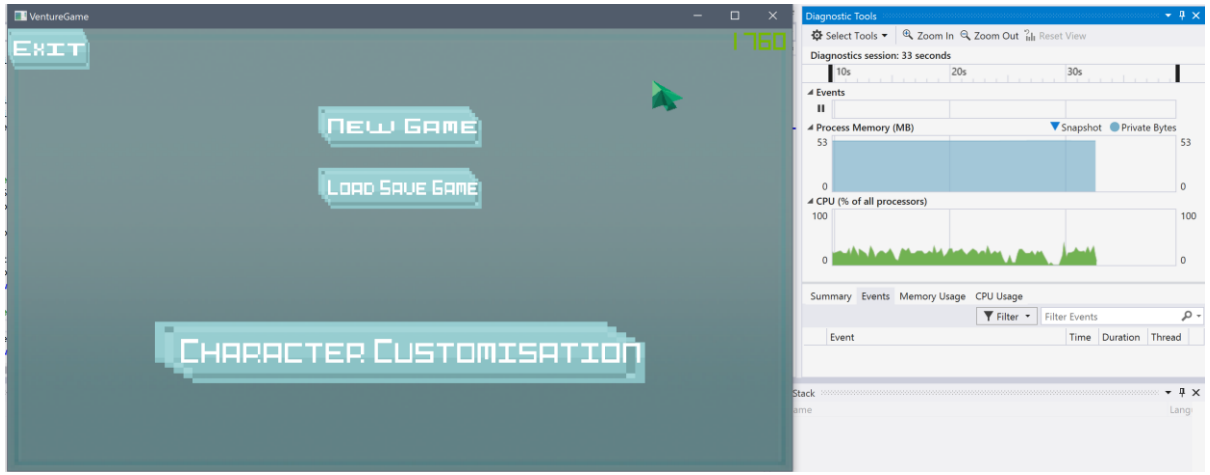
However this does start to dip when I open the players Inventory. The first thing I will start to implement is a time out for the inventory, because currently the inventory window flashes open and closed when you hold the button down.
Furthermore every time the inventory window is opened it has to refresh the icons in the window, so to start of I will just create a time out for the inventory, so it can only be opened every half a second or something.

Optimisation process restart, with some new game features and fixes.

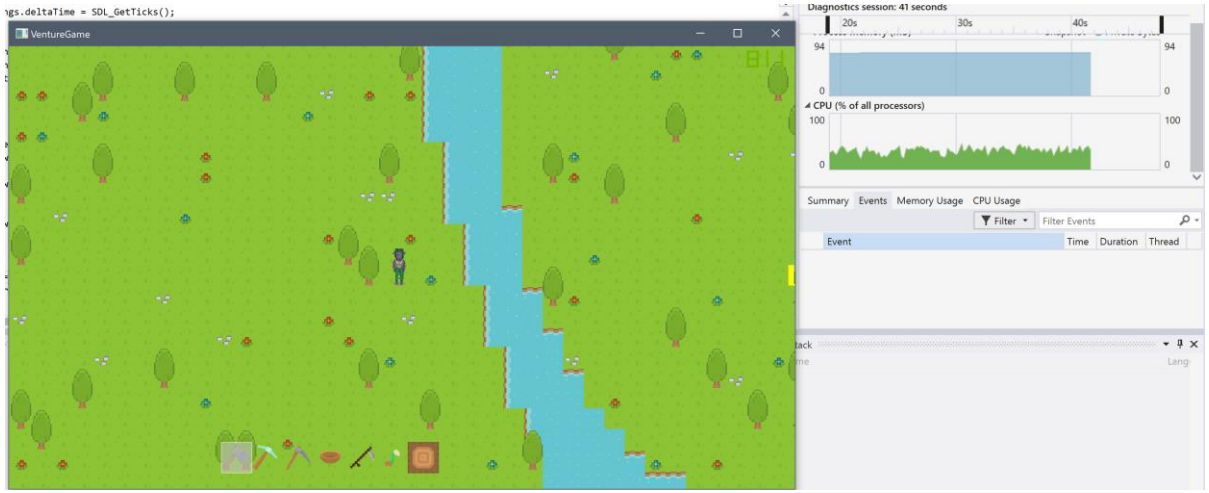
Over the Easter break, I dedicated some time to fixing old features that were currently not working, like pathfinding, saving/loading and character creation and rendering. So the results on the following slides are the game with these added features and fixes.

General Profiling Results

Now that the frame rate is normally over 60FPS, the SDL_RENDERER_PRESENTVSYNC flag that is being passed into the SDL window stops the game from rendering more frames than the refreshrate of the monitor, so to test the raw performance of the game, and not wait for VSYNC, I passed in SDL_RENDERER_ACCELERATED which does not limit the frame rate.

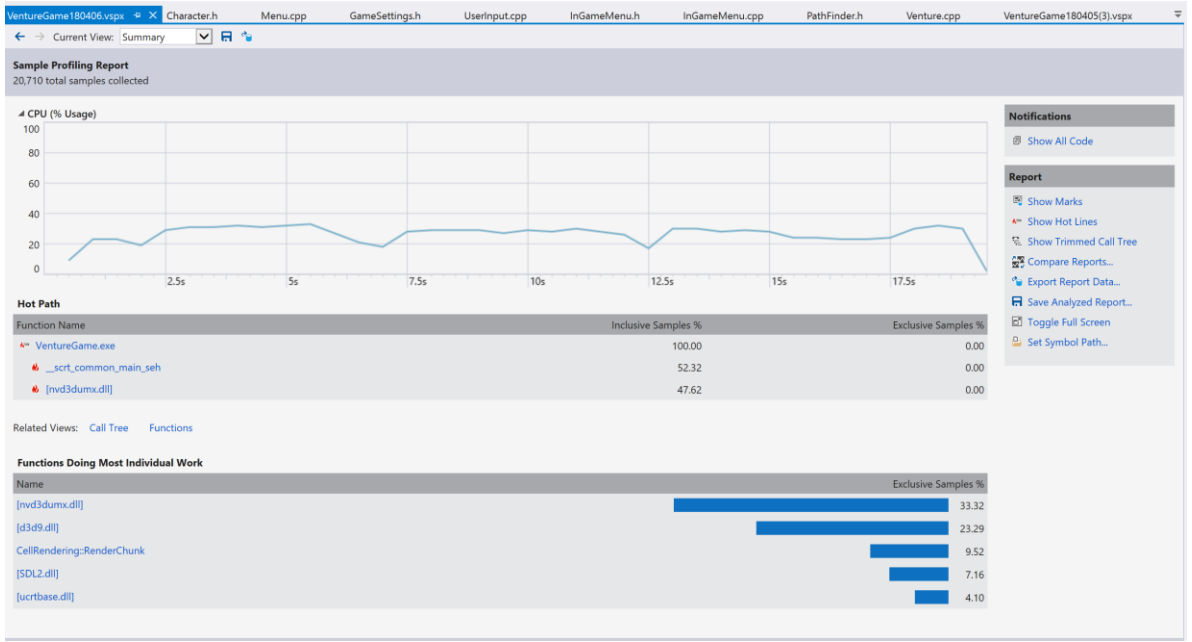


On the Menu screen of the game (above) the game uses at most 25% of the total CPU, this is probably due to the game only being single threaded, and my computer has 4 processing cores.



Even in game, the processor doesn't go much above 25% utilisation. One big optimisation I could do would be to make the game mult-threaded, and offload some of the other calculations to another core.

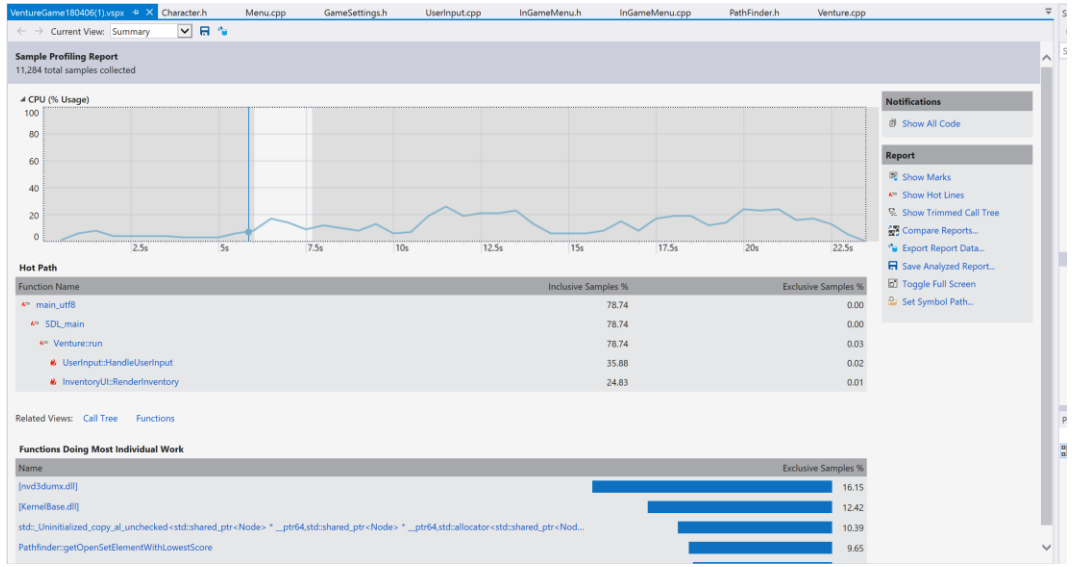
Render Chunk is the only function in the hot path that is using a lot of processing power, the other functions are SDL/openGL.



Render Chunk is the only function in the hot path that is using a lot of processing power, the other functions are SDL/openGL.

Stress Testing Game Features:

In previous benchmarks, I have not been focusing on performance of individual components, mainly just what uses the most processing for general walking around game play. However when specific components are overused by the player, i.e pathfinding or adding and removing lots of items from the inventory, this requires a lot of processing.



This is the CPU usage of when I was constantly adding and removing items from the inventory and pathfinding around the player

Handle user input is the function that calls pathfinding, so I wills start with optimising that.

Pathfinding:

The pathfinding works in this game, by offsetting the search area from the start and end point, so that the area being searched is in the positive X and Y area of the level, this was due to not being able to have negative values in vectors.

The current pathfinding in this game searches an area of 100x100 around the start point.

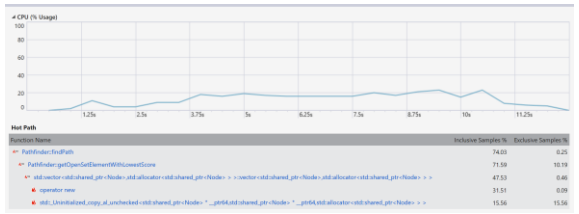
```
// Create nodes for every cell in the grid
for (int x = 0; x < searchSize; x++)
{
    nodes.push_back(std::vector<std::shared_ptr<Node>>(searchSize, nullptr));
}
```

To optimise this I will start by trying to dynamically scale the search area to the distance between the start and end points.

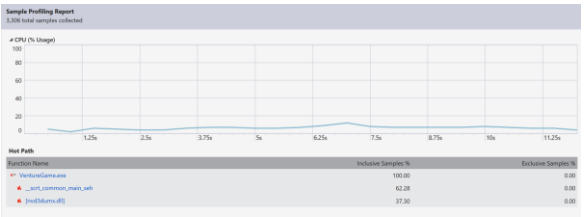
```
//change searchSize based on distance between target
searchSize = euclideanDistance(start, goal) * 2 + minSearchSize;
std::cout << "SearchSize: " << searchSize << std::endl;
if (searchSize <= 0)
    searchSize = minSearchSize;

// Create nodes for every cell in the grid
for (int x = 0; x < searchSize; x++)
{
    nodes.push_back(std::vector<std::shared_ptr<Node>>(searchSize, nullptr));
}
```

This simple code will reduce the amount of nodes that need to be created for searching a small area. Thus the function needs to spend less time creating the area to search, when it isn't necessary to create so many nodes.
The drawback is that there isn't a guaranteed path that it will be able to get, due to the limitations of the search area being based of the distance between the two points.

Before



After



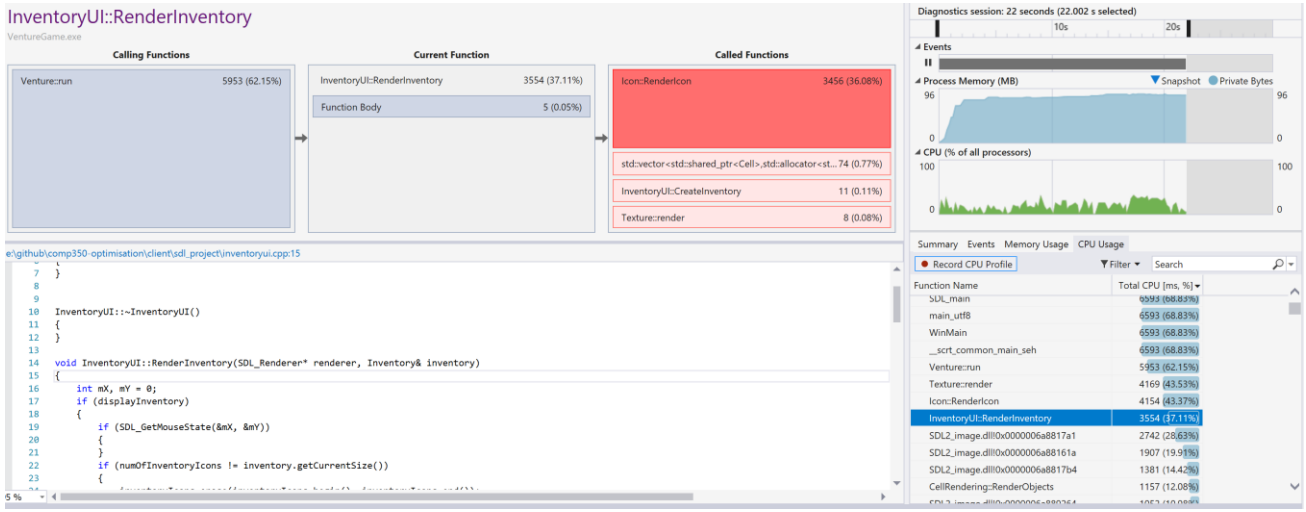
Pathfinding:

The size of the area of nodes being generated for each A* search is represented by the large black square.

Image of the pathfinding search area and path, after the change.

Inventory:

Currently when the inventory is open, and the player collets or removes an item, the inventory is re-creating itself, which makes the game lag for a short while as it re-creates everything.



This bit of code checks if the size has changed, and if it has, it will refresh the inventory by deleting everything and re-create it. This is obviously really inefficient, and when I made it I was suppose to refactor it later, but never did.

```
if (SDL_GetMouseState(&mX, &mY))
{
}
if (numOfInventoryIcons != inventory.getCurrentSize())
{
    inventoryIcons.erase(inventoryIcons.begin(), inventoryIcons.end());
    CreateInventory(renderer, inventory);
    numOfInventoryIcons = inventory.getCurrentSize();
}
else
{
    backgroundTexture.alterTransparency(150);
    backgroundTexture.render(renderer, getX(), getY(), getWidth(), getHeight());
```

This bit of code checks if the size has changed, and if it has, it will refresh the inventory by deleting everything and re-create it. This is obviously really inefficient, and when I made it I was suppose to refactor it later, but never did.

Inventory:

Old code, which simply deleted and created the inventory every time an item was added or removed.

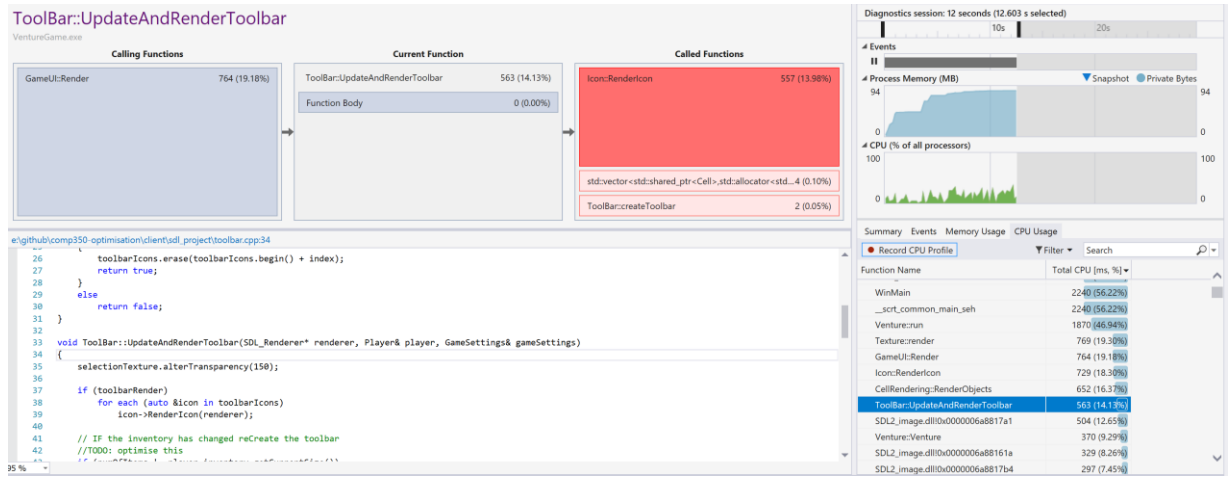
```
// Creates and populates the inventory UI
void InventoryUI::CreateInventory(SDL_Renderer* renderer, Inventory& inventory)
{
    // Set the starting position for the icons to render
    int x = getX() - getWidth() / 2 + iconSize;
    int y = getY() - getHeight() / 2 + iconSize * 2;

    // loop through all the inventory items and create the icons
    for (int i = 0; i < inventory.getCapacity(); i++)
    {
        Icon icon;
        auto sharedIcon = std::make_shared<Icon>(icon);

        if (x > getX() + getWidth() / 2 - iconSize)
        {
            x = getX() - getWidth() / 2 + iconSize;
            y += iconSize;
        }
        sharedIcon->setX(x);
        sharedIcon->setY(y);
        sharedIcon->setWidth(iconSize);
        sharedIcon->setHeight(iconSize);
        sharedIcon->renderBackground = true;
        inventoryIcons.push_back(sharedIcon);
        x += iconSize;
    }

    // Set the icons
    for (int i = 0; i < inventory.getCurrentSize(); i++)
        inventoryIcons[i]->setIconItem(inventory.get(i));
}
```

The old code, which simply deleted and created the inventory every time an item was added or removed.

New code, doesn't erase or create any items, just sets the value of the items to the same as the players inventory. Furthermore it only renders the current size, rather than the maximum size of the inventory.

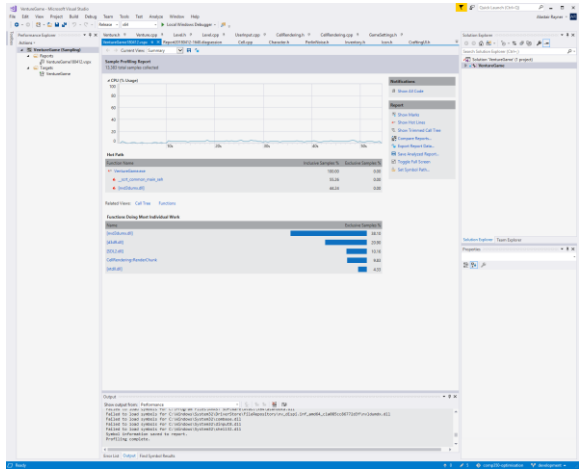
```
// Render all the icons in inventory
for (int i = 0; i < inventory.getCurrentSize(); i++)
{
    if(inventoryIcons[i]->getIconItem().type.Resource == NULL)
        inventoryIcons[i]->setIconItem(inventory.get(i));

    if (inventoryIcons[i]->getIconItem().type.Resource != inventory.get(i).type.Resource)
        inventoryIcons[i]->setIconItem(inventory.get(i));
    inventoryIcons[i]->RenderIcon(renderer);
}
```

It's hard to show in the profiler the results of this optimisation in the profiler, however now when anything is added or removed from the inventory, the game doesn't judder and the inventory doesn't flicker.

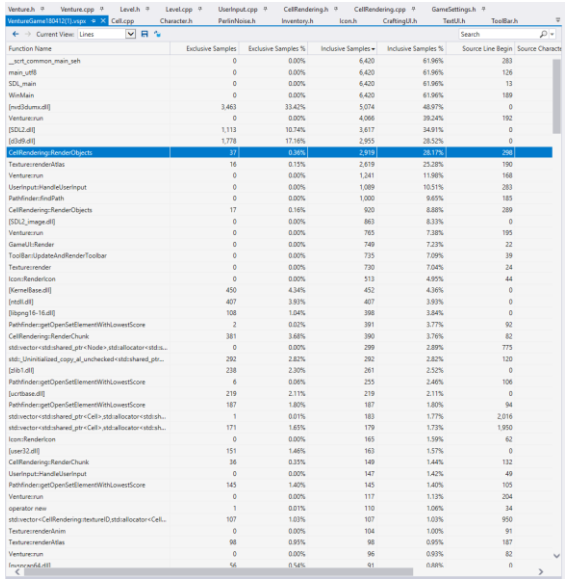


General Profiling:

When profiling all these results, I generally have windows task manager open aswell, as it is often a good tool to quickly identify bottlenecks.



When running in release mode, the Render Objects and render Atlas functions both are still the two largest resource hogs, using 2,919 and 2,600 inclusive samples. Compared to pathfinding which is using 1000.



The profiler has inclusive and exclusive samples, Inclusive samples is the amount of execution time spent in a function, including the time spent in functions that the function calls.

Exclusive samples is the time spent in that function, excluding the time spent in functions in that function.

Concurrent Chunk Processing:

One of the ways I tried to tackle the optimisation of the cell rendering function, was to make it multi threaded. My idea was to make each chunk be able to process and render the cells inside it concurrently.

However as there are so many dependencies for the rendering of cells, this made it extremely difficult, and ended up creating more problems than it solved.

```
std::vector<SDL_Thread*> threads;
// Render all the cells in the chunks
for (int i = (camera.getX() / level.getCellSize()) / level.getChunkSize() - 1; i < ((camera.getX() / level.g
    for (int j = (camera.getY() / level.getCellSize()) / level.getChunkSize() - 1; j < ((camera.getY() / lev
    {

        SDL_Thread* thread = SDL_CreateThread(t_RenderChunk, "Chunk Thread", (void *)&level.World[i][j]);
        threads.push_back(thread);
    }
```

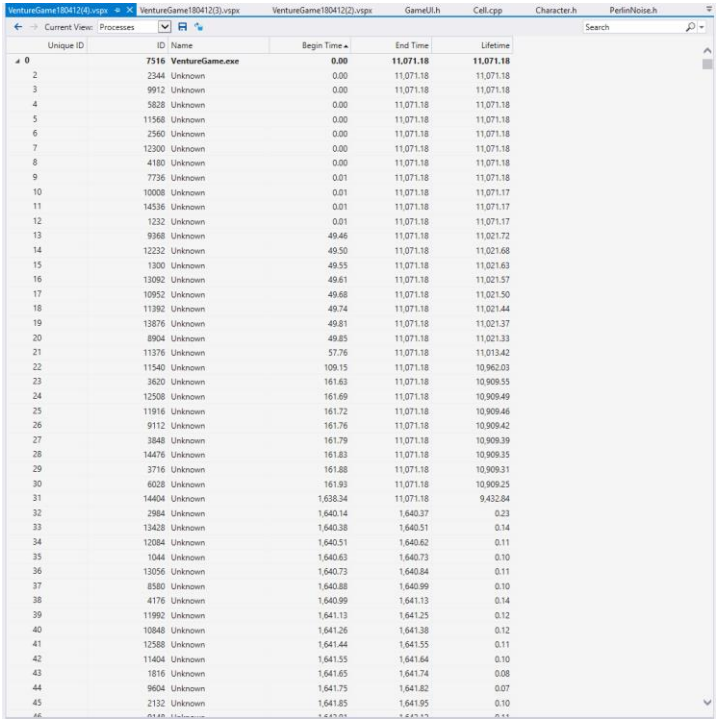
I ended up not implementing these changes as it decreased the performance and created a few bugs with the game.

The reason for why this decreased performance is due to it was creating a bunch of threads each frame, and each thread didn't have much processing to do. All it was doing was rendering about 38 chunks, which had 8x8 cells.

```
for each (auto& thread in threads)
{
        SDL_WaitThread(thread, NULL);
}
```

One improvement I will try to make in the future is to make the whole cell rendering class run in it's own thread instead of creating a thread for each chunk. However the render function has a lot of dependencies that I will have to work around, which will take a lot of work.
As I have found multi-threading in C++ is significantly harder than when I did it in java.

This is the performance difference from when I tried to do threading for my program. As I was creating and deleting so many threads, this lead to significantly decreased performance, as you can see on the right, all the threads being created as the time increases.
There were roughly 50,000 threads created for a 10 seconds game.



Unique ID	ID	Name	Begin Time	End Time	Lifetime
0	7516	VentureGame.exe	0.00	11,071.18	11,071.18
2	2344	Unknown	0.00	11,071.18	11,071.18
3	9912	Unknown	0.00	11,071.18	11,071.18
4	5828	Unknown	0.00	11,071.18	11,071.18
5	11568	Unknown	0.00	11,071.18	11,071.18
6	2560	Unknown	0.00	11,071.18	11,071.18
7	12300	Unknown	0.00	11,071.18	11,071.18
8	4180	Unknown	0.00	11,071.18	11,071.18
9	7736	Unknown	0.01	11,071.18	11,071.18
10	10008	Unknown	0.01	11,071.18	11,071.17
11	14536	Unknown	0.01	11,071.18	11,071.17
12	1232	Unknown	0.01	11,071.18	11,071.17
13	9368	Unknown	49.46	11,071.18	11,021.72
14	12232	Unknown	49.50	11,071.18	11,021.68
15	1300	Unknown	49.55	11,071.18	11,021.63
16	13092	Unknown	49.61	11,071.18	11,021.57
17	10952	Unknown	49.68	11,071.18	11,021.50
18	11392	Unknown	49.74	11,071.18	11,021.44
19	13876	Unknown	49.81	11,071.18	11,021.37
20	8904	Unknown	49.85	11,071.18	11,021.33
21	11376	Unknown	57.76	11,071.18	11,013.42
22	11540	Unknown	109.15	11,071.18	10,962.03
23	3620	Unknown	161.63	11,071.18	10,909.55
24	12508	Unknown	161.69	11,071.18	10,909.49
25	11916	Unknown	161.72	11,071.18	10,909.46
26	9112	Unknown	161.76	11,071.18	10,909.42
27	3848	Unknown	161.79	11,071.18	10,909.39
28	14476	Unknown	161.83	11,071.18	10,909.35
29	3716	Unknown	161.88	11,071.18	10,909.31
30	6028	Unknown	161.93	11,071.18	10,909.25
31	14404	Unknown	1,638.34	11,071.18	9,432.84
32	2984	Unknown	1,640.14	1,640.37	0.23
33	13428	Unknown	1,640.38	1,640.51	0.14
34	12084	Unknown	1,640.51	1,640.62	0.11
35	1044	Unknown	1,640.63	1,640.73	0.10
36	13056	Unknown	1,640.73	1,640.84	0.11
37	8580	Unknown	1,640.88	1,640.99	0.10
38	4176	Unknown	1,640.99	1,641.13	0.14
39	11992	Unknown	1,641.13	1,641.25	0.12
40	10848	Unknown	1,641.26	1,641.38	0.12
41	12588	Unknown	1,641.44	1,641.55	0.11
42	11404	Unknown	1,641.55	1,641.64	0.10
43	1816	Unknown	1,641.65	1,641.74	0.08
44	9604	Unknown	1,641.75	1,641.82	0.07
45	2132	Unknown	1,641.85	1,641.95	0.10

Comparison complete.

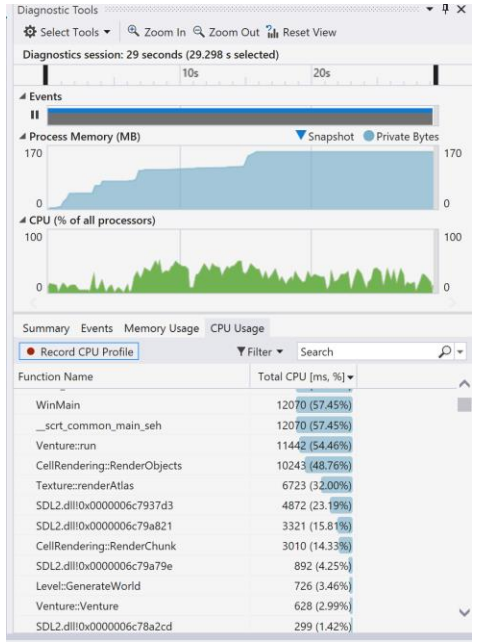
Comparison Column	Delta	Baseline Value	Comparison Value
Menu::MainMenu	7.78	3.02	10.80
[SDL2_image.dll]	7.46	4.34	11.81
Menu::CharacterCustomisationMenu	6.42	2.38	8.80
Player::RenderPlayer	5.57	3.31	8.87
Texture::renderAnim	5.54	3.26	8.80
[nvd3dumx.dll]	4.96	12.02	16.98
[d3d9.dll]	4.31	7.19	11.50
[libpng16-16.dll]	3.34	1.60	4.94
Player::renderCharacterBody	3.34	1.76	5.09
[win32u.dll]	2.47	4.47	6.94
Texture::render	2.31	1.01	3.32
[zlib1.dll]	2.23	1.40	3.63
Button::render	2.11	0.83	2.93
Player::renderCharacterClothes	1.64	1.06	2.70
TextUI::render	1.59	0.57	2.16
[user32.dll]	1.13	0.72	1.85
ToolBar::UpdateAndRenderToolbar	1.10	0.44	1.54
WinMain	-1.35	71.03	69.68
main_utf8	-1.35	71.03	69.68
SDL_main	-1.35	71.03	69.68
__scrt_common_main_seh	-1.35	71.03	69.68
Level::GenerateWorld	-1.42	2.04	0.62
Venture::run	-1.47	69.99	68.52
[SDL2.dll]	-4.26	72.32	68.06
CellRendering::RenderObjects	-9.08	63.32	54.24
[ucrtbase.dll]	-9.38	50.50	41.13

The image to the left shows the decreased performance, of renderObjects, compared to a previous profiler run.

(Ignore the increase in menu performance, this is due to me having the menu open for a longer period than the previous run)

Chunk Object Pooling:

The problem:



As the player explores the level, the world generates chunks of tiles.
This process is costly as the game creates the Chunk object every while the player is moving, which can cause a slight judder as the chunk object is being created.

Moreover the objects never actually get deleted, so the std::map of objects continually grows, which is okay if the player doesn't explore very far, but if they this is a kind of memory leak, as the memory of the game will just continually increase.

Possible solution:

To try and fix this issue, I will create a timer on each chunk object, which is refreshed if the chunk is still on screen.
However when the chunk is offscreen and the chunks timer goes above a set threshold, the chunk will be added to the pool of empty chunks, which when the CreateChunk function is called, the chunk will be gathered from the pool of unused chunks.

The solution:

I created a ReplaceChunk function that erases the tiles and recreates the new tiles, overwriting where the old, unused chunk data was.
It is currently adding the unused chunks to a vector of points, after the chunk is out of the camera view for 10 seconds.

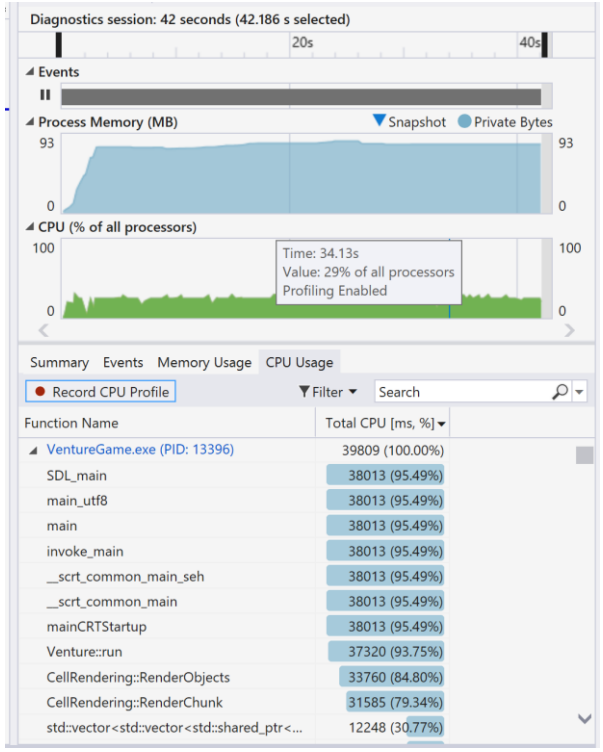
```
// Generates a hashmap of chunks around the camera
void Level::GenerateWorld(Camera& camera)
{
    // Calcualte chunks in camera area
    int numOfChunksWidth = ((camera.WindowWidth / cellSize) / chunkSize) + levelGenerationRadius;
    int numOfChunksHeight = ((camera.WindowHeight /cellSize) / chunkSize) + levelGenerationRadius;
    camera.ChunksOnScreen.x = numOfChunksWidth;
    camera.ChunksOnScreen.y = numOfChunksHeight;
    int numOfChunksGen = 0;
    // loop through the chunks and create or replace the chunks
    for (int i = ((camera.getX() / cellSize) / chunkSize) - levelGenerationRadius; i < ((camera.getX() / cellSize) / chunkSize) + numOfChunksWidth; i++)
    {
        for (int j = ((camera.getY() / cellSize) / chunkSize) - levelGenerationRadius; j < ((camera.getY() / cellSize) / chunkSize) + numOfChunksHeight; j++)
        {
            // If the chunk hasnt already been created
            if (World[i][j] == NULL)
            {
                // Create the chunks if there are no inactive chunks
                if (unusedChunks.size() <= 0)
                    CreateChunk(i, j);
                // Replace an active chunk
                else
                    ReplaceChunk(i, j);
                proceduralTerrain.populateTerrain(World[i][j]);
                numOfChunksGen++;
            }
            // If the chunk has been out of the camera area for a set time
            if (World[i][j]->activeTime.getTicks() > chunkTimeout && World[i][j]->isActive())
            {
                unusedChunks.push_back(std::make_pair(World[i][j]->getX(), World[i][j]->getY()));
                World[i][j]->setActive(false);
            }

        }
    }
```

This code loops through the chunks and then creates, replaces or adds it to a list of unused chunks

Chunk Object Pooling:

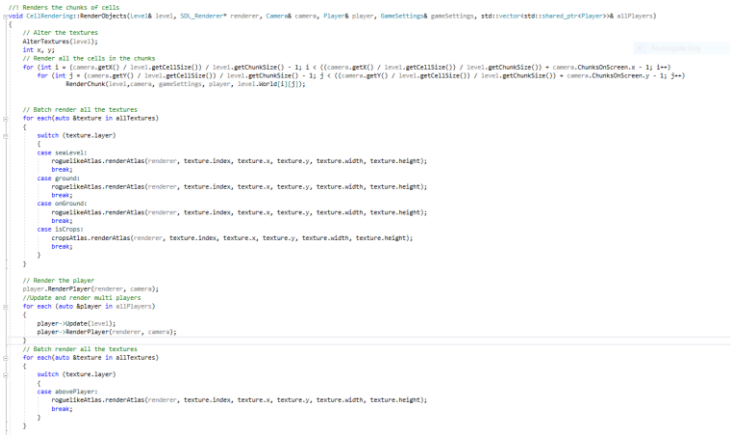
Checking:

Now when the player explores the level, the chunks that the player has been to, but is no longer in camera range, will get replaced with any new chunks data that gets created.
This shows a great improvement to memory performance, as the memory useage doesn't constantly increase as the player explores the level, and instead stays level, with the new chunk data overwriting the old data.



Small Batch Rendering Change

When I was looking over the rendering, I realised I was looping through a large vector twice, this could be resolved by creating two different vectors, of different sizes.
Because currently I had one vector of all the textures locations in the level, which I was looping over once before I rendered the player, and then again to render over the player. This means there was a lot of loops that were not actually doing anything.



To rectify this I simply created two different vectors, and when I added the texture, checked which vector it should go in with a simple if else statement.

```
//! A vector of all textures
std::vector<textureID> texturesBelowPlayer;
std::vector<textureID> texturesAbovePlayer;
```

```
void CellRendering::AddToBatchRendering(int ID, int x, int y, int& size, char layer)
{
    textureID texture;
    texture.index = ID;
    texture.x = x;
    texture.y = y;
    texture.width = size;
    texture.height = size;
    texture.layer = layer;

    if(layer < layers::isCrops)
        texturesAbovePlayer.push_back(texture);
    else
        texturesBelowPlayer.push_back(texture);
}
```

This is such a small change, however because there were so many textures, which had to be looped over each frame, it had a significant impact.

When comparing the profiler from before and after the change, the performance was significantly greater.

Comparison complete.

Comparison Column	Delta	Baseline Value	Comparison Value
mainCRTStartup	7447	0	7447
main	7447	0	7447
__scrt_common_main	7447	0	7447
invoke_main	7447	0	7447
main_utf8	6897	550	7447
__scrt_common_main_seh	6897	550	7447
SDL_main	6897	550	7447
Venture::run	6811	547	7358
CellRendering::RenderObjects	6542	416	6958
CellRendering::RenderChunk	6403	75	6478
std::vector<std::vector<std::shared_ptr<Cell>,std::allocator<std::shared_pt...	2519	0	2519
std::vector<std::shared_ptr<Cell>,std::allocator<std::shared_ptr<Cell> >...	2516	0	2516
std::vector<std::vector<std::shared_ptr<Cell>,std::allocator<std::shared_pt...	1593	0	1593
std::vector<std::shared_ptr<Cell>,std::allocator<std::shared_ptr<Cell> >...	1570	0	1570
std::_Vector_alloc<std::_Vec_base_types<std::shared_ptr<Cell>,std::allocato...	886	0	886
std::_Vector_alloc<std::_Vec_base_types<std::vector<std::shared_ptr<Cell>,...	860	0	860
std::_Vector_alloc<std::_Vec_base_types<std::shared_ptr<Cell>,std::allocato...	688	0	688
std::_Vector_alloc<std::_Vec_base_types<std::shared_ptr<Cell>,std::allocato...	655	0	655
std::_Vector_alloc<std::_Vec_base_types<std::vector<std::shared_ptr<Cell>,...	654	0	654
std::_Vector_alloc<std::_Vec_base_types<std::vector<std::shared_ptr<Cell>,...	644	0	644
std::_Vector_alloc<std::_Vec_base_types<std::vector<std::shared_ptr<Cell>,...	642	0	642
std::_Vector_alloc<std::_Vec_base_types<std::shared_ptr<Cell>,std::allocato...	638	0	638
CellRendering::AddToBatchRendering	508	0	508
std::vector<CellRendering::textureID,std::allocator<CellRendering::texturel...	492	0	492
std::_Vector_alloc<std::_Vec_base_types<std::shared_ptr<Cell>,std::allocato...	478	0	478
std::vector<CellRendering::textureID,std::allocator<CellRendering::texturel...	471	13	484
std::_Vector_alloc<std::_Vec_base_types<std::vector<std::shared_ptr<Cell>,...	447	0	447
std::shared_ptr<Cell>::operator-><Cell,0>	445	0	445
std::_Compressed_pair<std::allocator<std::shared_ptr<Cell>...	423	0	423
std::shared_ptr<Chunk>::operator-><Chunk,0>	402	0	402
std::_Compressed_pair<std::allocator<std::vector<std::shared_ptr<Cell>,st...	399	0	399
std::vector<CellRendering::textureID,std::allocator<CellRendering::texturel...	370	0	370
Level::GenerateWorld	277	7	284
std::_Compressed_pair<std::allocator<std::shared_ptr<Cell>...	224	0	224
std::_Compressed_pair<std::allocator<std::vector<std::shared_ptr<Cell>,st...	220	0	220
std::_Ptr_base<Cell>::get	204	0	204
std::_Ptr_base<Chunk>::get	175	0	175
std::_Vector_alloc<std::_Vec_base_types<CellRendering::textureID,std::alloc...	154	0	154
ProceduralTerrain::populateTerrain	147	4	151
std::_Vector_const_iterator<std::_Vector_val<std::_Simple_types<CellRender...	131	0	131
ProceduralTerrain::generateGround	127	4	131

Comparison complete.

Comparison Column	Delta	Baseline Value	Comparison Value
VentureGame.exe	82.61	9.39	92.00
dwmapi.dll	-1.35	1.38	0.03
libpng16-16.dll	-1.48	1.64	0.16
zlib1.dll	-1.89	2.24	0.35
user32.dll	-2.08	2.50	0.42
ntdll.dll	-3.16	3.45	0.29
KernelBase.dll	-5.25	5.94	0.69
d3d9.dll	-8.04	8.44	0.40
SDL2.dll	-8.64	9.39	0.75
win32u.dll	-16.58	17.66	1.08
nvd3dumx.dll	-30.78	33.07	2.30

Conclusion

Throughout this optimisation process, I have been focusing on the large scale improvements to the algorithms within the game, rather than micro optimisations.

However throughout this process I have been doing micro optimisations for variables, i.e. using char or short, instead of int or double, where appropriate. This is not mentioned in the report, because I wasn't sure how to document all the minor changes, but they are available on my github fork.

Almost all the changes made in this game were for the CPU, as SDL manages most of the GPU stuff for you, moreover the GPU was not the bottleneck throughout the process, and the main issue with my code, was that it is only single threaded, however I struggled with implementing C++ multi-threading, and when I finally got it working, there was a decrease in performance, I decided to work on other optimisations I could do instead.

There were also a few memory issues with the game at the start, such as when you explore the world, none of the data gets deleted, so the memory usage will constantly increase till it crashes. However I fixed this by having a timeout for each chunk, so as the player explores, chunks they visited a while ago get re-used as new chunks.