Sprite Sorting and Camera Culling

By Bosco Barber

Index

1. Introduction

First things first
Let’s see why do we need it...

- Dynamic sorting system
- Render optimization
- Sorting optimization
- Performance improvement

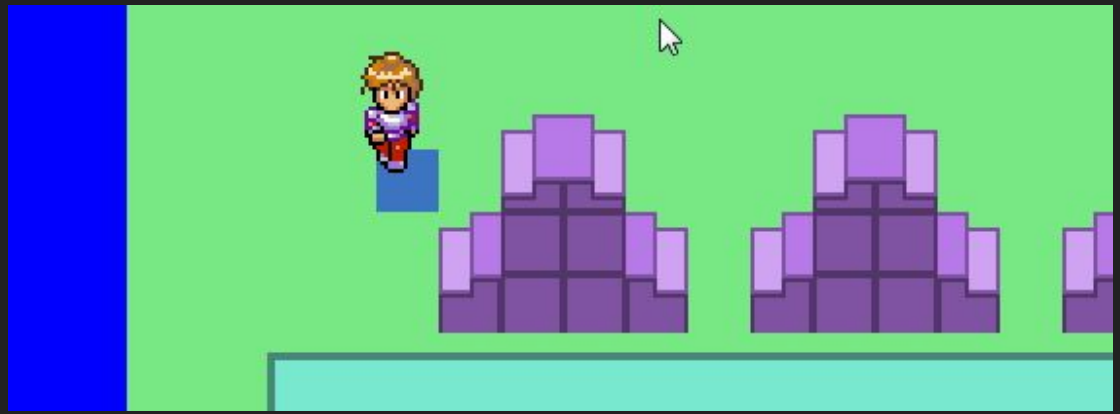
2. Sprite Sorting

2.1. What is sprite sorting?

- Right from the beginning, the idea was born out of the desire to represent reality.
- To represent a 3D world in 2D.
- Transition from side and overhead (perpendicular) view games to 2.5D, top-down (¾) and isometric view games.
- Sprite sorting is needed to represent depth.

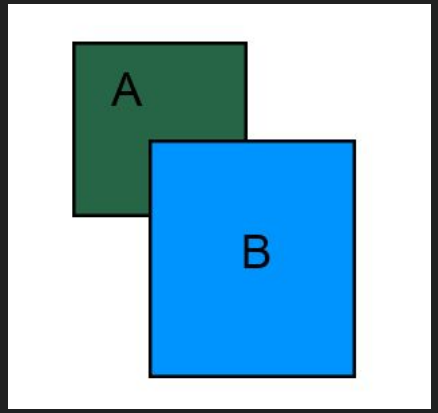
2.2. Why is sprite sorting so important?

- Illusion of depth.
- Coherence with the perspective of the environment.
- Automatic sprite sorting system. Avoid doing it manually and hardcoded.



2.3. Z-order as a concept

- Ordering of overlapping 2D objects
- Variable that will determine which sprite has to be printed above.

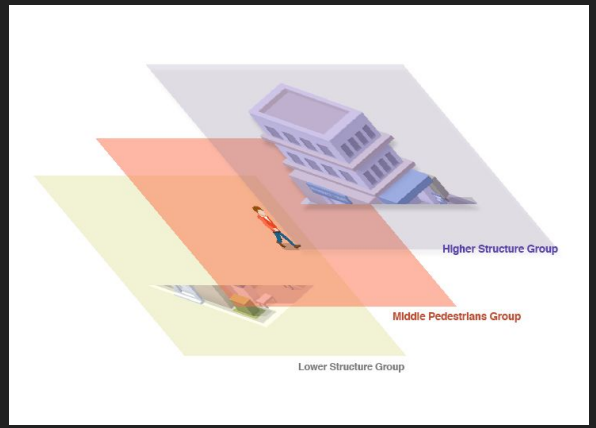


2.4. Different approaches by different games

There are some systems to sort sprites, it depends on the type of game, the resources of the machine and the code structure.

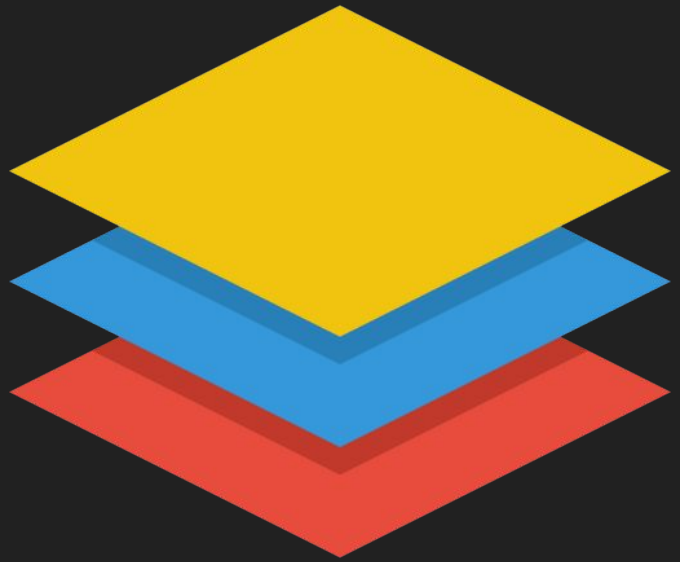
2.4.1. Cut sprites

- Separate sprites in two parts.
- Render first the lower part, later all the entities, and finally the higher part.
- This method is not automatic.
- Hard to maintain. It can get quite complicated with lots of objects.
- We could get in trouble with other moving objects or entities.



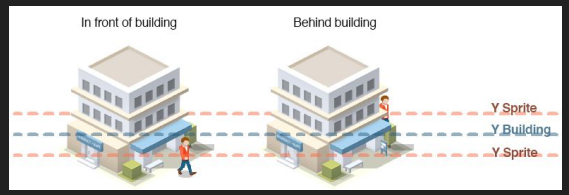
2.4.2. Sorting layers

This is the most common system used, but it could be used with different approaches.



2.4.2.1. By Position

- All entities and objects will be sorted by Y position.
- From the lower Y to the higher Y. From top of the window to down.
- Entities placed higher will be rendered before entities placed lower.
- We may need to take into account the height of the sprite or another offset. To do that, we can use a pivot.
- That could consume more resources than we expected, because we must sort a lot of objects.



2.4.2.2. By Colliders

- It’s not the ideal way to sort sprites, and it is not widely used.
- For complex structures.
- Based on colliders to change entities’ layers and activate or deactivate colliders.

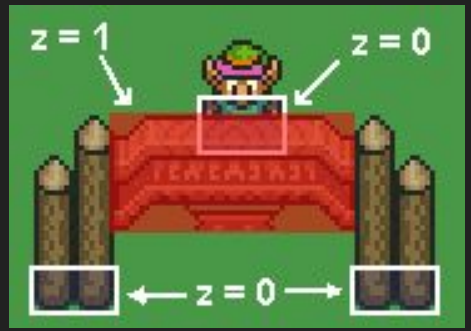
[Guinxu video]

2.4.2.3. By Vector 3D

- Common in isometric maps.
- The most difficult way to implement.
- We must use 3 dimensions.
- We will have to project the vector3 to 2D.

2.4.3. Complex objects

- For complex structures and objects with different heights.
- Assign a different Z value depending on how far from the ground is each object. “Z” has nothing to do with z-order in the context.
- Get all the objects with the same Z sorted separately.

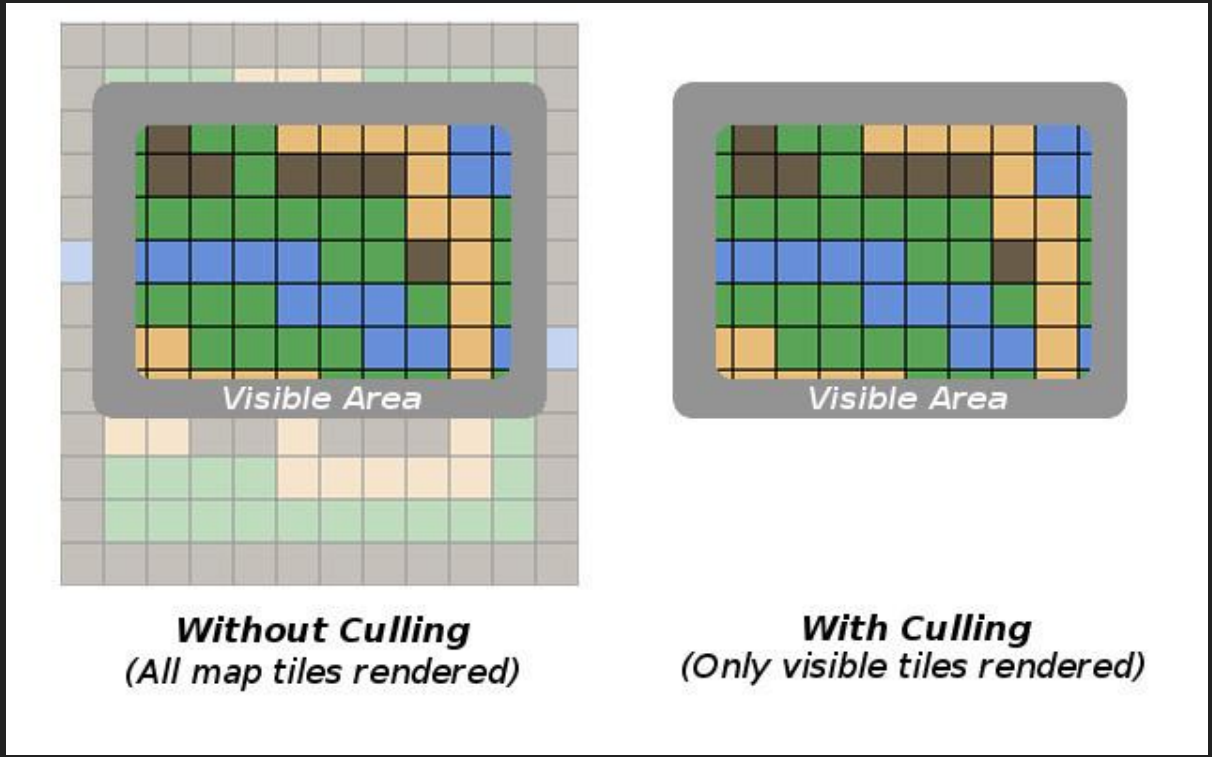



```
objects = {

link, (z = 0, lowY = 64)

column_a, (z = 0, lowY = 100)

column_b, (z = 0, lowY = 100)

topPart (z = 1, lowY = 80)

}
```

3. Camera Culling

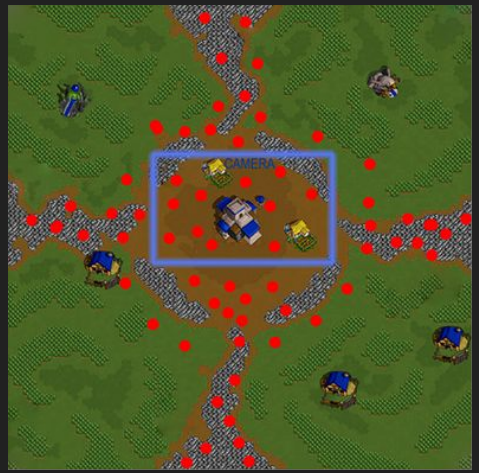
3.1. What is camera culling?

- Basic method that allows the program to only work with entities and objects that are on the camera viewport.
- Only render tiles and sprites that are on screen.
- Only manage and sort the sprites of the entities and the objects that are in the camera viewport.
- We need to check if what we are going to render is inside the camera or not.



3.2. Why is camera culling important?

- Load less sprites and entities each cycle.
- Save resources to the machine.
- Optimize the game.
- Increase performance.



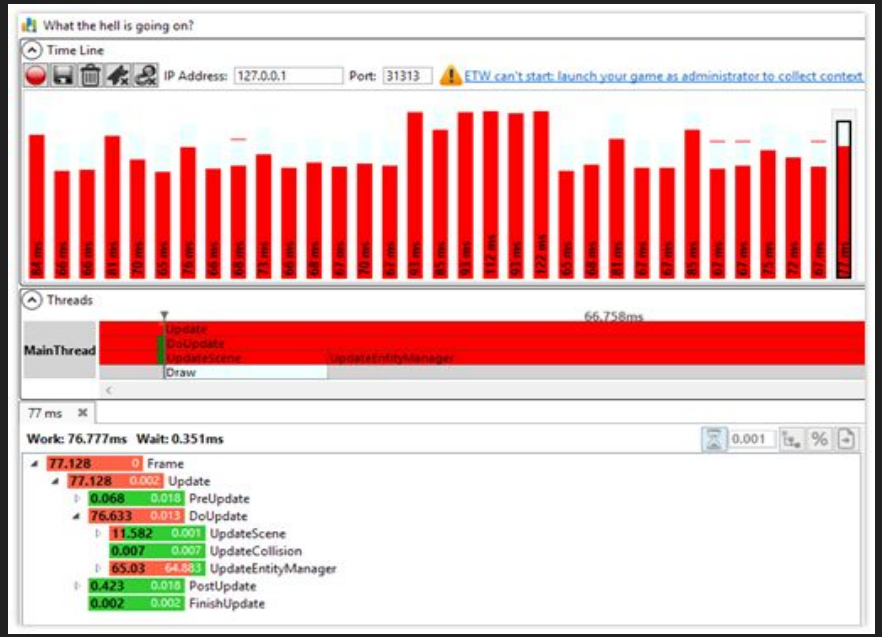
No Camera Culling

Camera Culling for entities

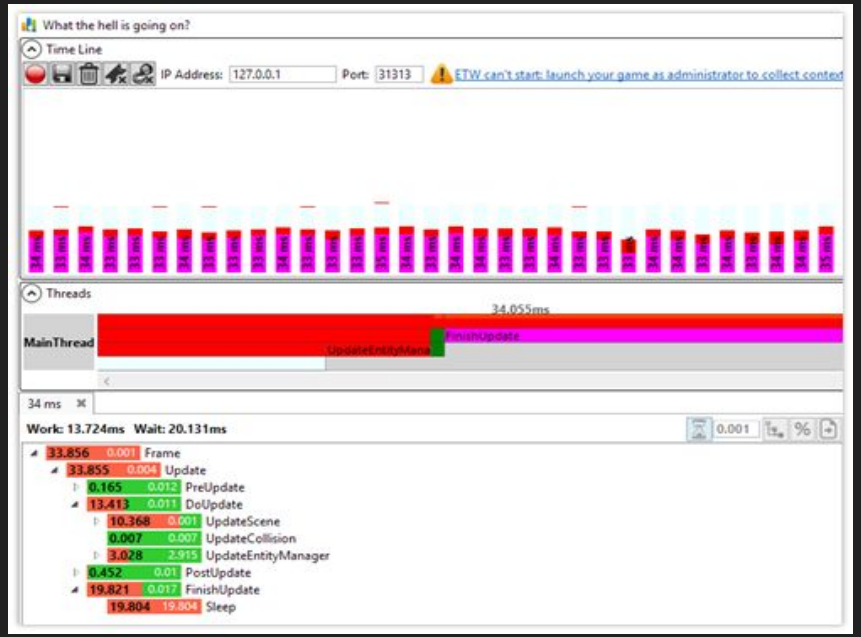
Camer Culling

3.2. Why is camera culling important?

Without



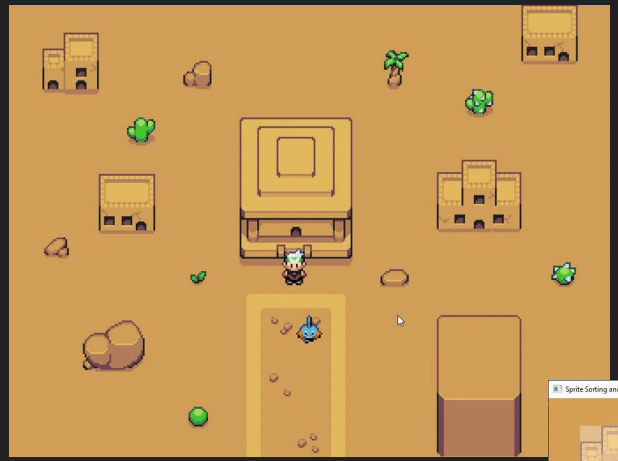
With

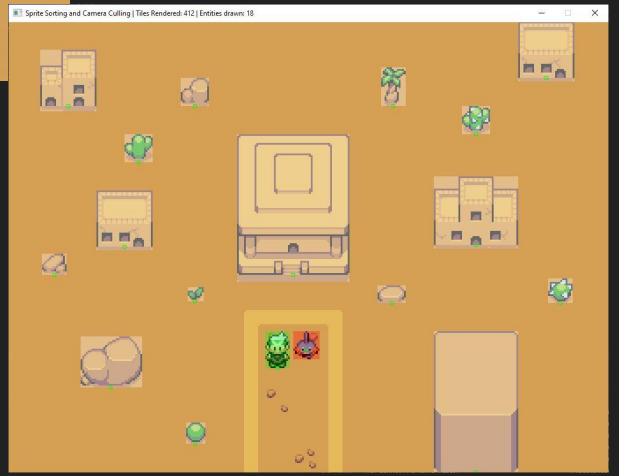


4. Selected approach

4.1. Overview

- We will sort sprites by (Y) position.
    - Pivot.
- We won’t operate with layers, we will operate directly with a list of entities.
- Tiled implementation.
- We will implement a camera culling to prevent rendering tiles that are not on the camera viewport. Moreover, it will only sort and render the sprites of the entities and objects on screen.





4.2. How do we implement sprite sorting in our game?

We want to have a list with all the sprites already sorted in order to render before entities above. To do that there is a function in the <*algorithm*> library to sort vectors. The function is *sort*.

In order to properly sort the list (vector), this function needs 3 parameters as arguments:

1. Beginning of a vector.

2. End of a vector.

3. A bool function that receives two elements of the vector as arguments and returns true or false depending on the priority we have implemented. This function will determine which element has to be placed before in the vector.

To create this function, we have to take into account the logic of our sprite sorting. In our case, it will depend on the Y position of our pivot. The logic of the function is actually very simple. It is created on *EntityManager::SortByYPos* and sorts the position of an entity.

```
std::sort(drawEntities.begin(), drawEntities.end(), EntityManager::SortByYPos);
```

4.3. How do we implement camera culling in our game?

To create a camera culling effect in our game is pretty simple, we just have to select what we want to render. To do that we are just going to add a filter to our rendering system. This filter will determine if something is outside the camera by checking if the rectangle of the element that we want to render intersects with the camera rectangle or not and once it has determined it, the sprite will be rendered or ignored.

In order to do that we will have to implement a filter similar to the next one and apply it to the render function or the function that pushes the elements into the list if you have done the sprite ordering system before:

```
if ((rect.x < -camera.x + camera.w && rect.x + rect.w > -camera.x) ||
(rect.x < -camera.x + camera.w && rect.x + rect.w > -camera.x))
{
     if (rect.y < -camera.y + camera.h && rect.y + rect.h > -camera.y) // Render
}
else // Don't render
```

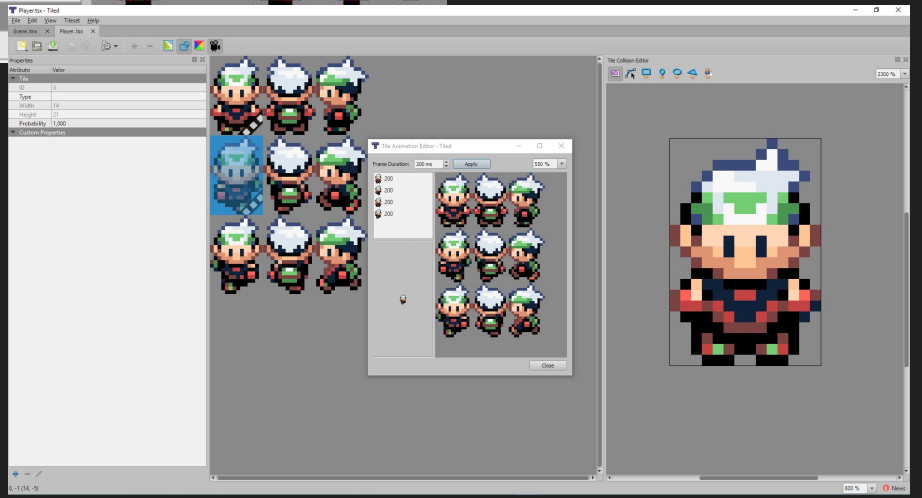
Anyway, as in this project we are using SDL, we will take advantage of it and in the implementation we will use the function *SDL_HasIntersection* that takes two rects as arguments and returns true if they intersect, or false if they don’t. Obviously, one rectangle will be the camera and the other the one of the element we want to render.

4.4. Working with Tiled

- Code to import entities and objects to the game.
- Load object layers in Tiled.
- Design our maps and place the entities directly in Tiled.
- Don’t worry if some object occupies different tiles!
- Easily managed by the sprite sorting system.

4.4.1. Importing dynamic entities from Tiled

- Easy to edit
- XML export
- Information
  - Variables
  - Animations
  - Colliders
- Integrated in base code



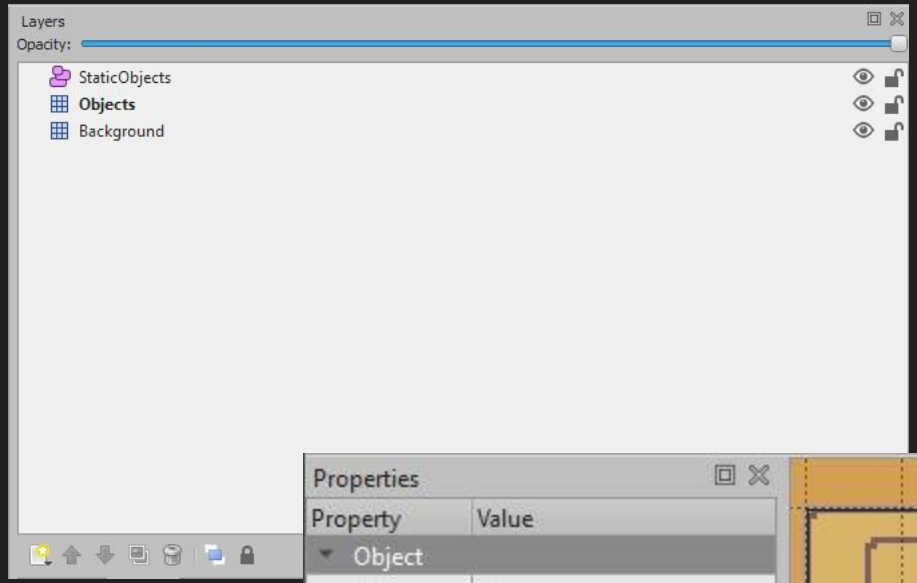
4.4.2. Importing static entities from Tiled

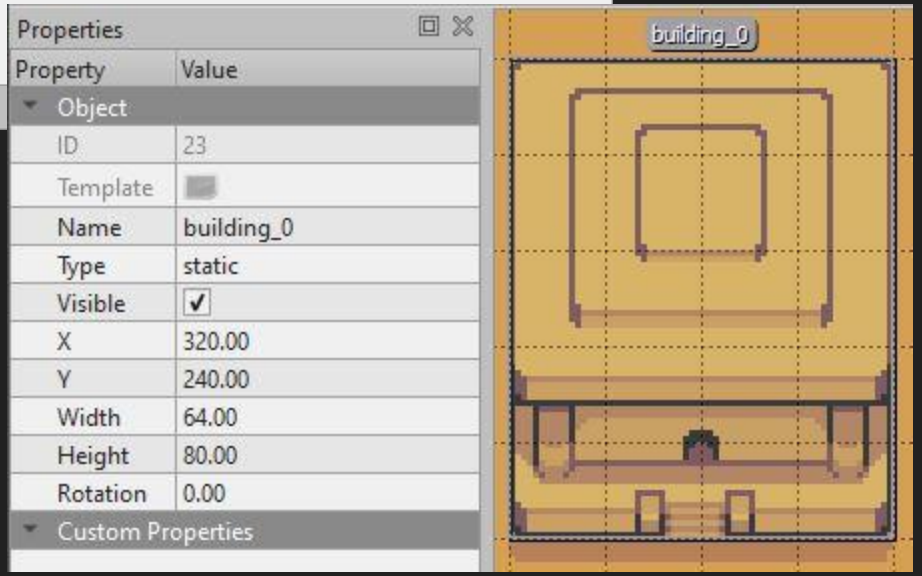
We will work with 3 layers:

1. **Background**
2. **Objects (paint)**
3. **StaticObjects (actual info)**

Important variables:

- **Name**
- **Type**





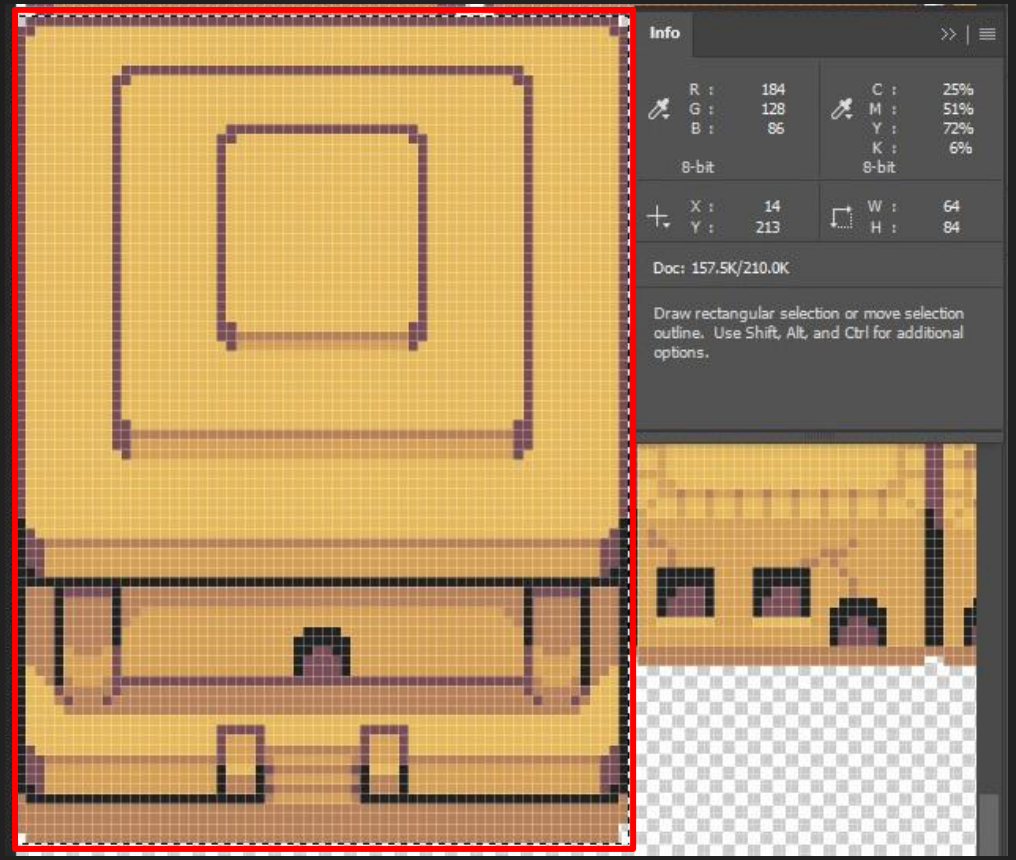
```cpp
//Load Object data ----------------------------------------------
pugi::xml_node objectGroup;
pugi::xml_node object;

for (objectGroup = mapFile.child("map").child("objectgroup"); objectGroup && ret; objectGroup = objectGroup.next_sibling("objectgroup"))
{
    for (object = objectGroup.child("object"); object; object = object.next_sibling("object")) {

        ColliderObject* obj = new ColliderObject();

        if (ret == true && object != NULL)
            ret = LoadObject(object, obj);

        data.colliders.push_back(obj);
    }
}
```

```cpp
bool Map::LoadObject(pugi::xml_node& nodeObject, ColliderObject* obj) {

    bool ret = true;
    if (nodeObject.empty()) ret = false;

    //Load Collider / Entity data
    obj->name = nodeObject.attribute("name").as_string();
    obj->entType = nodeObject.attribute("type").as_string();
    obj->tileId = nodeObject.attribute("id").as_uint();
    obj->collX = nodeObject.attribute("x").as_int();
    obj->collY = nodeObject.attribute("y").as_int();
    obj->collHeight = nodeObject.attribute("height").as_uint();
    obj->collWidth = nodeObject.attribute("width").as_uint();

    //Load Collider type from ObjectGroup
    pugi::xml_node objGroup = nodeObject.parent();
    std::string type(objGroup.child("properties").child("property").attribute("value").as_string());

    if (type == "COLLIDER_NONE")
    {
        obj->type = COLLIDER_NONE;
    }
    else if (type == "COLLIDER_FLOOR")
    {
        obj->type = COLLIDER_FLOOR;
    }

    return ret;
}
```

```cpp
void Scene::CreateEntities()
{
    //Iterate all objects of the map made with Tiled to find entities
    for (std::list<ColliderObject*>::iterator position = app->map->data.colliders.begin(); position != app->map->data.colliders.end(); position++) {
        if ((*position)->name == "player") {
            app->entities->CreateEntity(Entity::Types::PLAYER, (*position)->collX, (*position)->collY, (*position)->name);
        }
        else if ((*position)->entType == "static") {
            app->entities->CreateEntity(Entity::Types::STATIC, (*position)->collX, (*position)->collY, (*position)->name);

        }
        else if ((*position)->entType == "NPC") {
            app->entities->CreateEntity(Entity::Types::NPC, (*position)->collX, (*position)->collY, (*position)->name);
        }
        else {
            LOG("There isn't any entity with name %s and type %s", (*position)->name.data(), (*position)->entType.data());
        }
    }
}
```

```cpp
Entity* EntityManager::CreateEntity(Entity::Types type, int PositionX, int PositionY, std::string name)
{
    static_assert(Entity::Types::UNKNOWN == (Entity::Types)3, "code needs update");
    Entity* ret = nullptr;
    switch (type) {
    case Entity::Types::PLAYER: ret = new Player(PositionX, PositionY, name); break;
    case Entity::Types::STATIC: ret = new Static(PositionX, PositionY, name); break;
    case Entity::Types::NPC: ret = new NPC(PositionX, PositionY, name); break;
    }
    if (ret != nullptr) {
        entities.push_back(ret);
        ret->Start();
    }

    return ret;
}
```

```cpp
Static::Static(int x, int y, std::string name) :Entity(Types::STATIC, x, y, name)
{
    //Assign type of static entity, texture rect and pivot
    //Orthogonal map ------------------------
    if (name == "building_0") {
        type = Static::Type::BUILDING;
        SetRect(0, 144, 64, 84);
        SetPivot(32, 80);
    }
    else if (name == "building_1") {
        type = Static::Type::BUILDING;
        SetRect(64, 176, 32, 34);
        SetPivot(16, 32);
    }
```



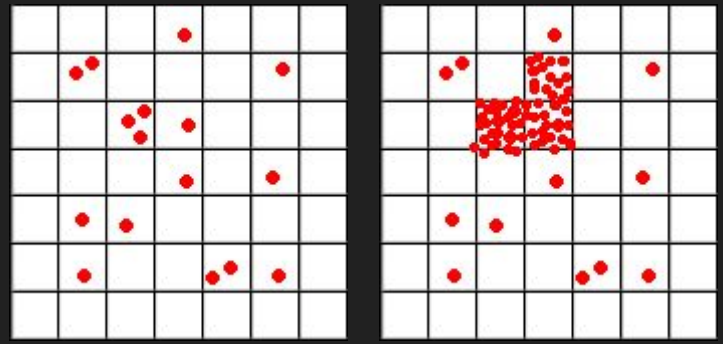
5. Possible improvements

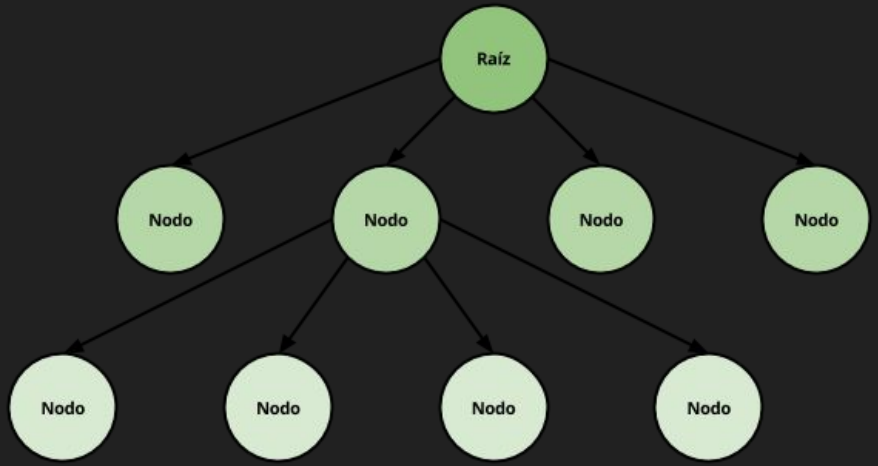
5.1. Quadtree

- At this point the camera culling system is not going to be really efficient because we are going to be checking if every element on the scene is inside of the camera.

- Space partitioning
  - Divide the space in small cells.
  - **Problem:** a lot of entities in the same cell.

- Quadtree
  - Type of space partitioning
  - To divide the cells into smaller pieces.
  - It's similar to a binary tree, but instead of two branches with **four branches** (child nodes).
  - Instead of having a static grid of nodes or cells, it generates automatically its own partitions.





5.2. Other improvements

- Loading textures is so expensive and now we have a texture loaded for every entity.
  - A possible solution is to have only the textures we will use, and set by an id the texture that we need.

- Sorting vectors with the *sort* function is expensive. There are a lot of methods and a lot of types of containers. Here you have some links to different types of sorting methods.
  - brilliant.org
  - geeksforgeeks.org

- Don’t iterate all the map, only from the beginning of the camera position to the camera position plus the viewport size.
  - Quadtree can be used here!

- Save all static entities in an XML file and load it.
  - We could also investigate a way to load pivot and frame of static entities in Tiled.

6. TODOs

TODO 1: Create *IsOnCamera* function

- You have to pass to the function a rectangle to determine if it is on camera or not. It is quite important to pass a rectangle, not only a position because tiles and objects have a width and a height.
- Returns true if something is on camera and false if it’s not.
- Camera moves in reverse of the other entities, change its sign.
- Scale is important. If there is something related with the scale of the pixels you have to put it to be able to know if it is real on camera or not.
- As we are using SDL, you could also use *SDL_HasIntersection*.
- Make use of const.

Test: You cannot test if it works until the next TODO.

TODO 2: Use previous function to only render tiles on camera

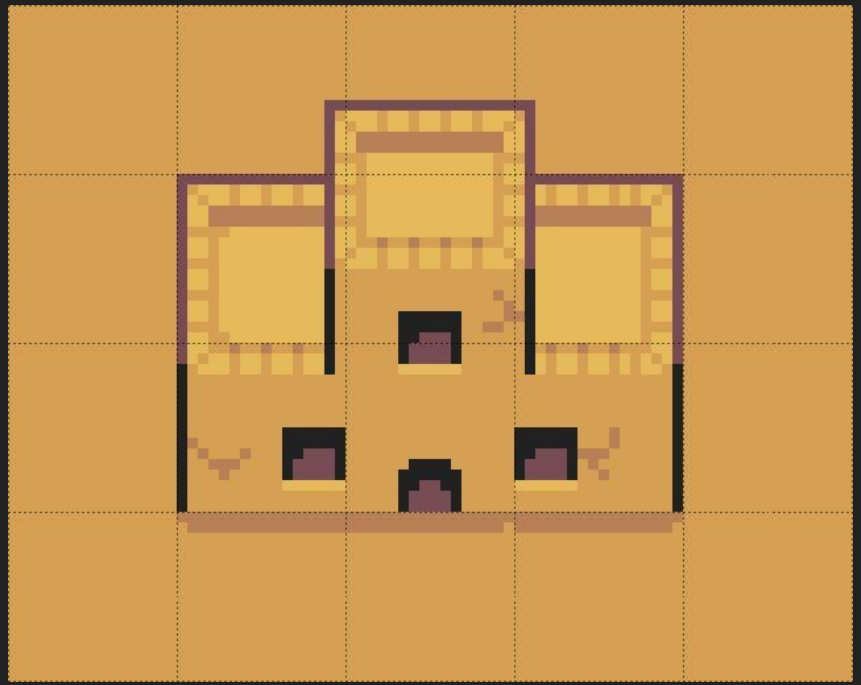
- You have to pass the position of the tile in pixels.
- And you also need the width and the height of the tiles of the map.
- Move with arrows and check that all is rendering well.
- Check in the title the number of tiles that are being rendered.
- Change the zoom level with 1, 2 and 3 keys and check that everything is fine.

Test: You can test it moving the camera in all directions and checking if the tile count on the title changes.

TODO 3: Create a building in Tiled and integrate it in code

- Remember name and type.
- Use Ctrl to create objects that fit in tiles.
- Texture rect (128, 169, 48, 41)
- Pivot (24, 39)

Test: In theory, if it has been well done, it will be where you put on the map. If it’s not, look carefully the steps and check if there is any LOG message.



TODO 4: Save entities that are on camera

- You have a vector called *drawEntities* that you must use to iterate. This vector will only contain the entities that are on camera at the time.
- During *Update* iteration you have to push back only entities on camera.
- Later, iterating that vector, you will have to draw entities.
- Move draw functions from main entities iteration to *drawEntities* iteration.

Test: You can move the camera out of the map and check if the entity count is 0 or not.

TODO 5: Sort entities

- To do that there is a function in the <*algorithm*> library to sort vectors. The function is called *sort*. Use *std::sort(Iterator first, Iterator last, Compare comp)* before iterating *drawEntities*.
- You have to pass the beginning and the end of a vector.
- You also have to pass a function. It is created on *EntityManager::SortByYPos* and sorts the position of an entity.
- Sort only the entities that are on camera. Remember that we have the vector *drawEntities* for that.
- Sort before *Draw*.

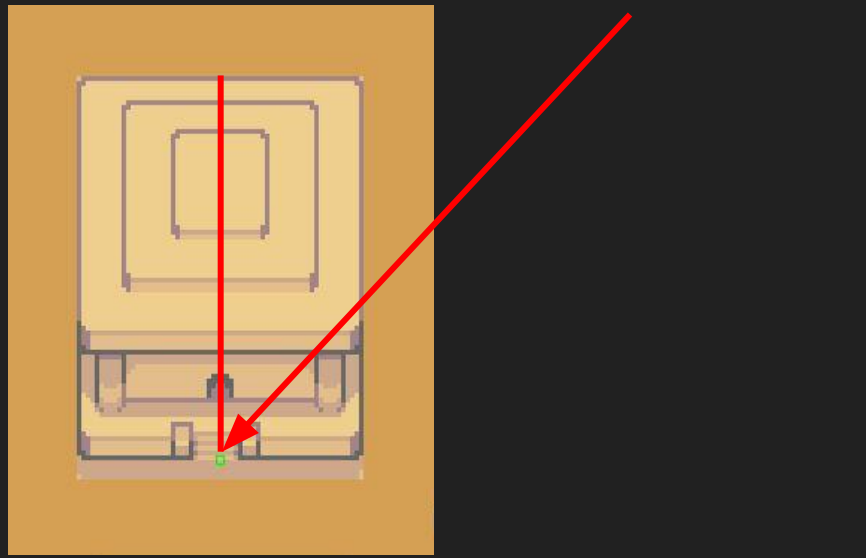
```
bool EntityManager::SortByYPos(const Entity* ent1, const Entity* ent2)
{
    return ent1->position.y < ent2->position.y;
}
```

Test: There will be some buildings and flora that will be correctly sorted. You can also move the player in the scene and check if it is working by position.

TODO 6: Add the pivot position to the sorting function

- Sort entities by pivot position. Just add the pivot y position to its entity.
- Pivot is a local position.

Test: Some objects will be sorted better with the player.



Congratulations!

Problems?

Check the solutions in the website!

7. Conclusions

Final thoughts...

- Game optimization
- Automated processes
- Multiple solutions
- Infinite improvements

Thank you for your attention!

Any questions?

GitHub: boscobarberesbert
Contact: boscobarberesbert@gmail.com