# Lauta Engine Technical Design Document

A guide to using the engine and its features

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Tuomas Suokko  
Version 1.0

# Table of Contents

[Lauta Engine Technical Design Document 1](#_Toc31294755)

[Table of Contents 2](#_Toc31294756)

[1 Getting started 4](#_Toc31294757)

[1.1 The main function 4](#_Toc31294758)

[1.2 The Application 4](#_Toc31294759)

[1.2.1 Global variables of the engine 4](#_Toc31294760)

[1.2.2 Global Manager variables 4](#_Toc31294761)

[1.2.3 InitEngine 5](#_Toc31294762)

[1.2.4 InitApplication 5](#_Toc31294763)

[1.2.5 Running 5](#_Toc31294764)

[1.2.6 HandleEvents 5](#_Toc31294765)

[1.2.7 Update 5](#_Toc31294766)

[1.2.8 Render 6](#_Toc31294767)

[1.2.9 Cleanup 6](#_Toc31294768)

[2 ECS 7](#_Toc31294769)

[2.1 The Basics of the ECS 7](#_Toc31294770)

[2.1.1 Manager 7](#_Toc31294771)

[2.1.2 Entity and Components 7](#_Toc31294772)

[2.1.3 System 8](#_Toc31294773)

[2.1.4 Group 9](#_Toc31294774)

[2.2 Creating a new Component 9](#_Toc31294775)

[2.3 Adding Components within Components 10](#_Toc31294776)

[3 Components within the Engine 12](#_Toc31294777)

[3.1 Movement components 12](#_Toc31294778)

[3.1.1 Transform 12](#_Toc31294779)

[3.1.2 Camera 12](#_Toc31294780)

[3.1.3 Input 12](#_Toc31294781)

[3.2 Graphical components 13](#_Toc31294782)

[3.2.1 Sprite 13](#_Toc31294783)

[3.2.2 Animation 13](#_Toc31294784)

[3.2.3 User Interface – UI Component 14](#_Toc31294785)

[3.2.4 UI Label 14](#_Toc31294786)

[3.3 Other Components 14](#_Toc31294787)

[3.3.1 Sound 14](#_Toc31294788)

[3.3.2 Collision 15](#_Toc31294789)

[3.3.3 Gameplay 15](#_Toc31294790)

[4 Asset Manager 16](#_Toc31294791)

[4.1. Textures 16](#_Toc31294792)

[4.1.1 Adding and using a texture 16](#_Toc31294793)

[4.1.2 Adding a texture with XML and using it 16](#_Toc31294794)

[4.2. Fonts 16](#_Toc31294795)

[5 Text Manager – UI Label 17](#_Toc31294796)

[5.1 The Script and Screenplay 17](#_Toc31294797)

[5.1.1 Adding text 17](#_Toc31294798)

[5.1.2 Editing existing text 17](#_Toc31294799)

[5.1.2 Drawing text 18](#_Toc31294800)

[6 Texture Manager 18](#_Toc31294801)

[6.1 Loading a texture 18](#_Toc31294802)

[6.2 Drawing a texture 18](#_Toc31294803)

[7 Event Manager 19](#_Toc31294804)

[7.1 Creating and subscribing to an event 19](#_Toc31294805)

[8 Tiled Support 19](#_Toc31294806)

[8.1 The Board Class 19](#_Toc31294807)

[9 Lauta Math 20](#_Toc31294808)

[10 OpenCV and the Dice Capture system 20](#_Toc31294809)

[11 Tips using the Engine 21](#_Toc31294810)

[12 Plans regarding improving the Engine 21](#_Toc31294811)

[13 Credits and references 21](#_Toc31294812)

# Getting started

Lauta Engine is an open source C++ game engine with an SDL-framework. Currently it can only be used for simple 2D games for PC. This is because, the engine cannot currently perform complex shading, realistic physics, or mobile development. It was specifically designed for creating virtual board games and has all the necessary components for that.

## The main function

When starting your new project, begin by downloading the engine from the git repository.

Most of your game logic will be created in the class “Application” and the main function will just loop through the applications *HandleEvents(), Update(),* and *Render()* functions in that order.

Before the loop the Application and the engine are initialized. The main function also has a frame rate cap on 60 fps and will regulate itself so that application will run as smoothly as possible with the same frame rate. For reference, the main function can be seen right here 🡪

## The Application

Here we’ll go through all the functions in Application and explain the purpose of each one is.

### Global variables of the engine

There are four global variables that are used throughout the engine. Global\_Event, Global\_Window, Global\_Renderer and Camera. Global\_Event is used primarily in the HandleEvents function used to poll SDL events. Global\_Window can be used to check window events, for example mouse click positions. Global\_Renderer is used to render everything in the application. Camera is used as a simple shortcut option if you want to add a moving camera component to the application.

### Global Manager variables

Three different managers are used to handle different gameplay elements and assets. “Manager” is used for the Entity Component System, where the Manager itself handles all the Entities in the manager. AssetManager handles all the assets that have been initialized for the application. This includes all the graphics, spritesheets, audio and fonts. EventManager is used for event-based triggers in the application. All the managers will be explained further in their own chapters.

### InitEngine

Here most of the SDL initialization happens. After we initialize everything related to SDL, which includes the global variables for the window, renderer and the added libraries of SDL font and audio systems. The most important of these are the *Globals::Global\_Window* and *Globals::Global\_Renderer*. If all of this is successful, the Boolean m\_IsRunning will be set to true, which will activate the while loop in main. If there is a need to add an SDL component or some other library, be sure to add the initialization in this function.

### InitApplication

In this function you can initialize all the gameplay components that are required for your project. This can include adding textures, fonts and sounds to the asset manager. After all of this you can create and initialize your gameplay objects.

### Running

This function just simply return the state of the Boolean m\_IsRunning. If the state of this Boolean is changed to false, for example in the function HandleEvents, this will quit the application.

### HandleEvents

In this function, SDL will poll any SDL Events from the global variable *Globals::Global\_Event*. Examples of SDL events are: SDL\_WINDOWEVENT, SDL\_MOUSEBUTTON, SDL\_KEYDOWN and SDL\_QUIT. These can be polled with a simple switch case checking *Globals::Global\_Event.type* .

//Wait for events and give them to the Global\_Event

SDL\_PollEvent(&Globals::Global\_Event);

//check what event is happening

switch (Globals::Global\_Event.type) {

case SDL\_QUIT:

m\_IsRunning = false;

break;

}

### Update

The Update and Render function are quite similar in that both update the game state and what’s happening in it. Update however must contain the **manager**’s *refresh* and *update* functions. This will affect everything under the Event Component System and go through all the components that have an overridden update function. The refresh function moves through all the entities and removes the ones that have been deleted prior.

If there is a component function within your entity that you wish to access, this can easily be done with the following piece of code:

(In this example, we will go through all the entities in the “ships” group and move their x-position using their Transform component.

for (auto& s : ships)

{

//move ships 1 x-unit forward

s->GetComponent<TransformComponent>().position.x += 1;

}

### Render

As said before, the same kind of update loops are used in the render function. Drawing entities will be used using the same update loop as shown above, but the drawing function is simple. The following SDL-functions need to be in the function, and all rendering must happen between them:

SDL\_RenderClear(Globals::Global\_Renderer);

// rendering between these lines

for (auto& s : ships)

{

//draw all ships

s->draw();

}

// nothing more to render

SDL\_RenderPresent(Globals::Global\_Renderer);

Render order is important, since the first entities that are rendered will be at the bottom of the rendering layer. Note that the SDL\_RenderClear function will start a memory leak if it runs while the window is minimized.

### Cleanup

In this function, this will just cleanup all the SDL Global variables and quit the application. You can use this for good memory management, especially if you are building an application for Linux.

# ECS

In this chapter we’ll go through the Entity Component System (ECS). There is a quick user guide in the header file of the ECS. The purpose of the ECS to categorize everything in your application for ease of use, adding only the necessary components to an entity and accessing those components when needed. The maximum number of components an entity can have is 32, but this can be increased, if for some reason this amount is exceeded.

## The Basics of the ECS

Wikipedia gives the following explanation why Entity Component Systems are used:

*“Every entity consists of one or more components which add behavior or functionality. Therefore, the behavior of an entity can be changed at runtime by adding or removing components. This eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain and extend.”*

With this brief overview, we can understand how the ECS in Lauta Engine works. Note that some may say this approach to an ECS that the engine uses, is more along the lines of an ECS-lite. This will be gone into detail in chapter 2.1.3.

### Manager

In short, the Manager keeps track of all the entities and components within a system. The manager class has two private vectors, one for all the entities and the other for all the groups. In the manager’s update() function, it will loop through all of the inherited update functions in all the components of the entity. The same is true for the draw() function. The Manager also has a refresh() function which will check if any entities in any group or in general have been removed from the system.

Adding an entity to your application can be done the following way:

Entity& Player(manager.addEntity());

Adding groups to your manager can also be done as simply as this:

auto& buttons(manager.getGroup(Application::groupUIButton));

This will be gone into further detail in chapter 2.1.5, but note that we use auto& as the variable, because readability, as it otherwise is   
 std::vector<Entity\*>& buttons(manager.getGroup(Application::groupUIButton));

For ease of use, you can have the entity as a global variable, but that might also be bad coding practice, so initialize it however you want. However, Manager should be a global variable in your application.

### Entity and Components

All components within entities have the same three overridden functions from the Entity class: init(), update(), draw(). When a components is added to an entity, it will call the init() function of that added components. The update() functions of the components within the entity are called when the manager itself calls its update function. The draw() function is mainly used in either the Sprite Component or just by calling a specific group’s draw function in the render() function of the application.

Adding a component to the entity is quite simple. Here’s an example of adding an animation component to an explosion effect:

Explosion.addComponent<AnimationComponent>(5, 140, false);

First, we write which entity we are addressing, then we call its addComponent function, having the desired component between the “< >”-brackets. Next in the round brackets we add the variables that are needed for the constructor of the component. In this example, the Animation Component requires 1. the number of frames in the animation, 2. the speed of the animation, and 3. if the playing state. An error will occur if you have a wrong amount or the wrong variables in the brackets.

If you want to check if an Entity has a specific component, you can call its hasComponent() function. For example, in the init() -functions of some Components, you can see a component check, that will add a desired component if that has not been added to the current entity. This example is from the Collider Component:

void init() override

{

if (!entity->hasComponent<TransformComponent>())

{

printf("add a Transform component to the collider entity!");

}

m\_transform = &entity->GetComponent<TransformComponent>();

}

The same idea is also in the hasGroup() function where we can check if an entity is a part of a specific group.

In that previous example we also used the GetComponent function, where we gave the entity’s Transform Component to the Collider Component’s m\_transfrom variable, that it needs to use to calculate collisions. GetComponent can also be used to retrieve functions or public variables within the components. For example:

UI.GetComponent<UILabel>().drawTextWithID("tutorial");

Other functions that can be used are the isActive() and destroy() functions. isActive checks if an entity is removed entirely by the destroy() function. These are mainly used in the manager’s refresh function which always checks the current state of all entities within the manager.

### System

The reason why I referred my ECS as an ECS-lite, is because the system part of the Entity Component System is not fully realized. This can however be easily expanded upon in the later future. Let’s say you would be working on an RPG, which would have a battle system, a text system, an inventory system and maybe an overworld system. As you can see, this can help the readability of your program and you can divide a project into different systems that teams can work on.

### Group

Groups are used for categorizing all components for ease of use for example in update of draw functions. To start using groups, first you need to set up an enumerator (named for example groupLabels) of size\_t, which can also be called “Group”. These should be set up in the header of the Application. As an example, we’ll have the following groups in our enumerator: groupUI and groupUIButton.

To loop through all the entities within a group, you need to set up a vector of entities. With our examples, we would have the following in the beginning of the Application:

auto& ui(manager.getGroup(Application::groupUI));

auto& buttons(manager.getGroup(Application::groupUIButton));

Adding a group to an entity is done the following way after you have set up your entity:

Entity& button(manager.addEntity());

button.addGroup(Application::groupUIButton);

Now that the buttons have been added to their group, we can use a for loop to go through all the entities within the group the following way:

for (auto& b : buttons)

{

//draw all buttons

b->draw();

}

If you would like to check a group ID using a switch case for example, you want use the following array for ease of use. This example would be for a Catan style game:

static const Application::groupLabels AllTiles[] = {

Application::groupOcean,

Application::groupTrees,

Application::groupWheat,

Application::groupBrick,

Application::groupSheep,

Application::groupDesert,

Application::groupSteel

};

## Creating a new Component

If you need a new Component for your application, creating one can be done the following way:

1. Create a new header for your new component class.
2. Include the header “Components.h”
3. Have this component subclass inherit the parent class “Component”
4. Create a default constructor
5. If you want to use the init(), update() or draw() functions, remember to override them.

## Adding Components within Components

In this example, we’ll be adding components within components. We’ll be using the component “Health Component” as an example. This is because, we want our health bar to have a position on screen, a sprite to draw it on screen and add it to a group for drawing purposes.

After we’ve followed the instructions on creating a new component, we then need to create a Struct for our health bar. This struct will have pointers to the components that the health bar uses and the entity itself. It will also contain the variables for its Group and Max Health amount.

The constructor of the struct will take a reference of the Component-Entity (I know this might be a bit confusing, but we’re treating our new component as if it was just an entity), our group ID and the max health number.

In our constructor we want address our variables to the referenced entity and get the needed components from it. An example of the class can be seen here:

struct HealthBar

{

TransformComponent\* trn;

SpriteComponent\* spr;

Entity\* ent;

Group gid;

int hp;

HealthBar(Entity& healthBar, Group groupID, int maxHP)

{

this->ent = &healthBar;

this->trn = &healthBar.GetComponent<TransformComponent>();

this->spr = &healthBar.GetComponent<SpriteComponent>();

this->gid = groupID;

this->hp = maxHP;

}

};

Now that we’ve created the struct, we can move onto our Health Component’s constructor. If you want multiple health bars for your entity, I suggest creating a vector of our created struct as a class variable. Otherwise, you want to use just one struct as a variable. My constructor will create three health bars, so I will be using the vector.

Then we can create a new entity and add it to the manager. Now, add the needed Transform and Sprite components and group ID to the entity. After all this, push back a Health Bar object to the vector with the entity, group ID and health amount, as stated above in the structs constructor.

It should look something like this:

std::vector<HealthBar> hb;

HealthComponent(int Health, Vector2Di positionOnScreen, std::string textureID,

std::string textureName, Group groupID)

{

for (int i = 0; i < 3; i++)

{

Entity& healthBar(manager.addEntity());

healthBar.addComponent<TransformComponent>(positionOnScreen.x,  
 positionOnScreen.y + 64 \* i);

healthBar.addComponent<SpriteComponent>(textureID,

AssetManager::Assets->getSourceRect(textureName),  
 1, SDL\_FLIP\_NONE, 0, false);

healthBar.addGroup(groupID);

this->hb.push\_back(HealthBar(healthBar, groupID, Health));

}

}

You can add a getter function for the entity (or the components) of your health bar struct using a function like this:

Entity\* getHealthBarEntity(int number)

{

return hb.at(number).ent;

}

Theoretically you could go even further beyond by adding components, within components, within components.

# Components within the Engine

In this chapter we’ll go through all the Components within the engine.

## Movement components

These components are mainly used to move entities around the screen.

### Transform

The transform component is used to set an XY-position for your entity. This is necessary if you want to have your entity appear on screen or have a collision trigger on your application. Anything that requires movement needs a transform component as well.

In its current form, there are three constructors for this component. One takes in two floats; x and y. The second takes a Vector2D, which is basically same, but more compact. The last constructor is the default, which takes no variables. This will set the entity’s position to x = 0 and y = 0, which is the upper left corner of the screen by default.

When the transform component is initialized, it will set the entity’s velocity to zero. This is because we don’t want the entity to move as soon as it comes into existence. You can however manipulate the Vector2D of velocity if you want movement for your entity.

In the update of the component, it adds the velocity multiplied by a speed variable to the entity’s position. This moves the entity and doesn’t move the entity if the velocity is zero.

Other functions that you can use are the getGlobalPosition() and getWindowPosition() functions. The global position refers to its default position variable, but window position takes into notice where the camera is currently located, so it subtracts that position from its position.

The initSize function is used in tandem with the Sprite Component, so this isn’t necessary to use outside the sprite size initialization. If you wish to manipulate the size of the entity, just change the scale value of the transform component.

### Camera

The camera component is very simple, as we’re manipulating the values of the Global Camera. When adding the component to the Camera entity that you create, it will take the following variables:

Vector2Di regarding in what area can the camera move around in, Vector2Di of its starting point and Vector2Di of UI dimensions, so that we wont scroll past the UI if there is one.

The component’s update function checks only if the camera’s movement is staying within its determined borders.

Otherwise, you can use the component to get the position of the camera with getCameraPos().

### Input

In its current form, the Input Component is not necessary for anything, as it was for testing purposes in the beginning of development. However, having inputs within the Handle Event function made things much simpler to develop for.

In the future, I wish to create a better and more intuitive version of the input component, that could be utilized for game development. This would include having options for different control types and assigning buttons and directional input for gameplay events.

## 3.2 Graphical components

These components are used to draw entities onto the screen.

### 3.2.1 Sprite

This component has multiple different constructors, so it can be a bit confusing.

The first one takes in a texture ID string that has been given to the asset manager, Vector2Di of the sprite’s size, Vector2Di of the position of the sprite in the sprite sheet it comes from, and the scale of the sprite.

The second one is mainly used alongside XML-sprite sheets, as it takes the texture ID, the source rectangle and the scale as variables. An SDL sourceRect is a struct that contains an x,y coordinate in the spritesheet, and width and height for sprite size.

The third constructor takes the same variables as constructor number two but takes the following extra variables as well: SDL\_RendererFlip for mirroring the sprite, int rotation for rotating the sprite and a bool for if the sprite should move on screen or stay at place with the screen. This is mainly used for UI purposes, for we don’t want our UI to move out of the way and stay in place.

The fourth constructor is identical to the third one, but this has the sourceRect separated into two different Vector2D variables, being the sprite size and the position on the sprite sheet.

All these constructors set the texture and retrieving the given texture ID from the asset manager.

When initialized, the sprite component retrieves the entity’s transform component and initializes the size using the transform component’s initSize()-function. After this, the sourceRect and the destinationRect of the texture is added.

During the update function, the component checks if the sprite should be moved with the camera, and if the scale of the sprite has been changed. This will move the destinationRect values accordingly.

The draw function is the main purpose of the sprite component, as it goes to the Texture Manager with its given texture, the sourceRect, destinationRect, flip state and scale, and draws the sprite on screen. You can read more about the Texture Manager in chapter 6.

Other useful functions within the component are the changeSourceTexture, where you can change the texture, if it’s on the same sprite sheet.

### 3.2.2 Animation

If you are using animations for your sprite, have all frames of the sprite in one sprite sheet file, going starting from left going right and down.

The constructor itself takes the following variables: int frames, int speed, and bool loop. Frames are self-explanatory, the speed of the animation is higher if the number is larger, and the loop bool just determines if the animation should loop immediately or not. The class contains an Animation struct, that saves the info of all these variables as an object.

The update function just goes through all the frames in the desired speed. This is lacking however the timing in where the animation frame is bound by the tick number, so your animation may start in the middle. This will be fixed in the future.

All animations are added to a map of animations, where each Animation object is paired with an ID. The constructor animation has the ID of “default”. When wanting to play an animation, you can call the function PlayAnimation with the desired animation ID. Adding an animation can be done with addanimation(), where the variables are an animation ID, frames and speed.

### 3.2.3 User Interface – UI Component

The UI component creates the UI from a tmx-file that was created with Tiled, giving the constructor the file path to the tmx-file, a group label and a scale int. The UI component uses the map creation and tmx-parsing of the Board class. The Board class return a vector of all the tiles created. This vector will be looped through giving them all the necessary components and group labels for drawing purposes. Note that all these UI tiles will be stationary.

### 3.2.4 UI Label

This component is used for the purpose of drawing text on screen using font files. This component takes a lot of variables for its constructor.

* Vector2Di position, where you want to place the text.
* String text, the text that you want to write.
* String font, the ID of the font that was added to the asset manager.
* String textID, an ID for the written text.
* Vector2Di text box size, how large of a text box you want.
* SDL\_Color, the rgb color of your text.

Going further into detail about the text manager will be done in chapter 5.

## 3.3 Other Components

These components are extra for making your applications a bit more complex and interesting.

### 3.3.1 Sound

The Sound Component has one constructor. It takes the following arguments: String ID, path to the music file and a Soundtype. The Soundtype is an enumerator, which can be either MUSIC or SOUNDEFFECT. Depending on which Soundtype was added, the ID and the file path will be added to the representing map, which will hold all the given soundfiles and their ID’s.

All audio will be added to their own channel. Background music only has one channel to use, because you don’t want to music channels playing at the same time. Sound effects all get their own channels to use. To play a sound effect, just call the playSoundEffect() function and give the sound effect ID as an argument. You can also choose to loop the sound effect, by giving the PlaySoundEffectOnLoop() function the ID and the amount of loops you want. If you wish to loop it infinitely, place a “-1” as the argument for loops.

Music itself can be paused, resumed and stopped with their corresponding functions.

### 3.3.2 Collision

The collider component’s constructor takes in by default only a string of a tag. This can be to identify what entities are colliding with each other. Otherwise, the initialization function gets the transform component of the entity and uses that to calculate collisions. The update function just moves the collider alongside the transform function.

Actual collision calculation happens within the Collision class. Here we currently have a simple AABB-styled collision detection, which only checks if the two sprites overlap, returning true or false.

The collider component although has the click function, which allows objects to become buttons. This will return an SDL mouse button event if the cursor is inside the entity, while also defining if it’s stationary, clicking down or ending the click.

### 3.3.3 Gameplay

This is where more gameplay specific components can be found for games designed like board games. It can also be a good comparison how to create more components for your application.

# Asset Manager

Before using the Asset Manager, be sure to add it into your Application file the following way:

AssetManager\* AssetManager::Assets = new AssetManager(&manager);

This way you’ll have a pointer to the Asset Manager itself while specifying which Manager you want to use alongside it.

## 4.1. Textures

When adding a texture, be sure to have the correct path to your texture file that you wish to use.   
I would suggest using the InitApplication() function to add all the assets used for the application.

### Adding and using a texture

Adding a texture is done the following way:

AssetManager::Assets->AddTexture("hex", "Assets/Spritesheets/HexTiles.png");

The first variable is the ID of the sprite that will be used later while drawing. The second variable is the source path to the image. Using the Sprite Component, just add the asset’s ID to the first variable when adding the component.

### Adding a texture with XML and using it

All the same applies what was written in chapter 4.1.1, but note that the current XML reading function can read only XML texture files that have been written the following way:

<TextureAtlas imagePath="Assets/Spritesheets/diceBlue.png">

<SubTexture name="dieWhite1.png" x="64" y="0" width="64" height="64"/>

<SubTexture name="dieWhite2.png" x="128" y="0" width="64" height="64"/>

<SubTexture name="dieWhite3.png" x="64" y="64" width="64" height="64"/>

<SubTexture name="dieWhite4.png" x="128" y="64" width="64" height="64"/>

<SubTexture name="dieWhite5.png" x="0" y="64" width="64" height="64"/>

<SubTexture name="dieWhite6.png" x="0" y="0" width="64" height="64"/>

<SubTexture name="dieCross.png" x="192" y="0" width="64" height="64"/>

<SubTexture name="dieGold.png" x="192" y="64" width="64" height="64"/>

</TextureAtlas>

The image path of the sprite sheet is taken from the XML-file. The subtexture’s name works as an ID, and the x, y, w, h variables are taken to size the image correctly from the sprite sheet image.

Later, when adding the XML texture to the entity and its Sprite Component, this should be written the following way:

AssetManager::Assets->AddTextureXML("ships","Assets/Spritesheets/ships.xml");

ship.addComponent<SpriteComponent>("ships",   
 AssetManager::Assets->getSourceRect("shipBlue.png"), 1.f);

Here, the texturesheetID is the given ID for the XML-file, and the textureName is the name ID that is referred in the Subtextures name.

## 4.2. Fonts

Fonts are added the following way:

AssetManager::Assets->AddFont("8bit", "Assets/Fonts/PrStart.ttf", 24 \* GLOBAL\_SCALE);

The first variable is the ID of the font, the second is the path to the font, and the last variable is the size of the font.

Writing something using the is done by using the UILabel component. An example of using the UILabel would be like this:

UI.addComponent<UILabel>(Vector2Di(100, 1500), "All other players have been defeated!",  
 "8bit", "victory", Vector2Di(650, 600), colorWhite);

Here the component’s variables are as follows:

1. Position of the text
2. The text that you wish to write
3. The ID of the font
4. The ID of the text
5. The size of the textbox
6. SDL\_Color variable

# Text Manager – UI Label

In this chapter we’ll go through how the text manager system works within the UILabel component. For an explanation of the constructor of the component, please refer to chapter 3.2.4.

## 5.1 The Script and Screenplay

For managing text, we need two map variables to hold all the info regarding the text. This is because we need something to identify the text and the graphical render of the text. The Script is a map that holds a string ID and a string for the given text. The Screenplay is a map that hold a string ID and an SDL Texture.

### Adding text

Adding text to the script and screenplay can be done with the function AddTextToScreenplay(). This function takes the following arguments: string textID, string text and string font. This function will not take a new position, as it by default will draw the text to the same position that was set in the constructor.

If you wish to add new text while giving it a new position, use the function AddNewTextWithNewPosition(). This takes the same arguments as before, but now takes a Vectro2Di position as well.

### 5.1.2 Editing existing text

Replacing a line of text but keeping the same ID can be done with the function replaceTextWithID(), giving the arguments string textID and string newText. There is however a slight memory bug regarding this because the deleted old text does not currently free memory. If using this, be careful of memory leaks.

If you wish to set your current text box position to a new one, use the function setNewPositionForText(), giving a Vector2Di as the argument. If you wish to move a specific line of text, use the function setPositionOfTextWithID().

### Drawing text

Drawing text can be done simple by calling the function drawTextWithID(), giving the wanted text’s ID tag.

# Texture Manager

The texture manager itself has two functions, loading a texture and drawing the texture. The sprite component’s draw function refers to the texture manager’s draw function. It is not suggested, but you can directly draw with the texture manager.

## 6.1 Loading a texture

A texture can only be used if it loaded into memory first. By using the texture manager’s LoadTexture() function, you only need to give the path to the texture file. PNG-files are suggested for use, for transparency and overall stability. The Asset Manager uses the loading function to add all the textures to its texture map. That way the sprite component can retrieve the wanted texture using only the textures ID and not its path.

## Drawing a texture

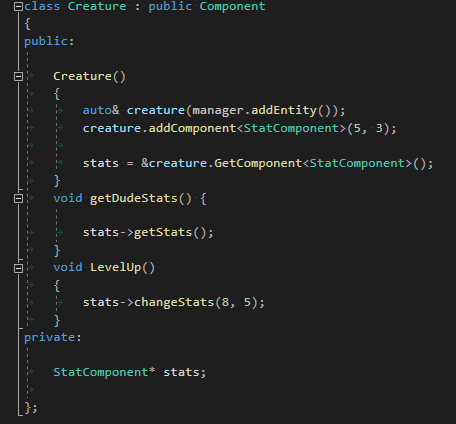
Drawing the texture itself we use the function Draw(), which just explands to SDL\_RenderCopyEx, which takes the Global renderer, the texture, a reference to the texture’s sourceRect and destinationRect, the angle of the image, and if the image should be mirror flipped. One of the arguments isn’t used, and therefor is just NULL, the center point of the image. The sprite component mainly uses this function, but if you wish to draw the texture directly, you can by calling the texture manager’s Draw() function.

# Event Manager

The event manager is a system, where you can create event-based triggers that activate functions. The idea is to have one function that many objects can subscribe to, and when that subscription is activated all object subscribed to that function will trigger.

Let’s say as an example we have an RPG where characters in a party can level up. Our level up function will be the one that will be triggered. Unfortunately these events can only be in the form of a void() without any parameters.

## 7.1 Creating and subscribing to an event

After you have initialized the event manager you can create an event with the following function:

EventManager \*myEventManager = EventManager::Instance();

myEventManager->createEvent("LevelUP");

We have now initialized an ID of “LevelUP” for our event. Now we need to attach our event to an Object and a function of the object. Let’s create a new class and an object called “creature”. With the created object, we can now attach the event ID to one of its functions, as shown below in the main function:

Creature \*dude = new Creature();

myEventManager->subscribe("LevelUP", dude, &Creature::LevelUp);

Now that our object has been subscribed to an event, we can now execute the event with it’s ID:

myEventManager->execute("LevelUP");

Now if we would check the stats before and after the execution, we would notice that the creature’s stats have been changed.

# Tiled Support

Tiled is a software created for the use of making 2D tile-based maps. In its current version there is a bug regarding using tile sheets. It is suggested, that you use tile sheets for your map that have spatial margins of 1 pixel. When setting up your map, have the map file and tile sheet file be in the same folder. It’s good to check in Notepad++ if your tmx-file image source is correct, just in case.

## 8.1 The Board Class

The board class is used for parsing through the .tmx file. It contains the Board class; a Tile struct and a TileSet struct. The TileSet is used to store the information of the tile set used to draw the map. The Tile struct holds the data regarding each tile in the created map. The Board class itself creates the map and returns a vector of Tiles. The UI component and its Map Component can be used to create the map itself.

tinyXML is used to extract and parse through the tmx-file, as it is just a glorified XML file.

# Lauta Math

In its current form, Lauta Math is used primarily for the Vector2D and Vector2Di classes. There are used to have coordinates be much more user friendly and readable, with vector calculation functions and debugging text.

There are a few mathematical algorithms as well, for checking if a point is within a boxed area, a random number generator, acquiring unit vectors and the angle of a given vector.

# OpenCV and the Dice Capture system

In my project demo I have added OpenCV functionality for a dice reading system for gameplay purposes. In this chapter, I will go through how this system works:

In our application, we first add the following includes:

#include "opencv2/imgproc.hpp"

#include "opencv2/highgui.hpp"

#include "opencv2/videoio.hpp"

#include <opencv2/features2d.hpp>

Then we can create a video capture object:

cv::VideoCapture cap(0);

With this created, we can then create a cv::Mat image, that will take the current capture frame and turn it into an image. Our unprocessed frame is then cloned and processed the following way:

cv::Mat Application::changeImage(cv::Mat image)

{

image.convertTo(image, -1, 1, 140); //brighten up the original image

cvtColor(image, image, cv::COLOR\_BGR2GRAY); //create a grayscale version of the image

blur(image, image, cv::Size(10, 10)); //blur the grayscale to reduce noise

threshold(image, image, 255, 255, cv::THRESH\_OTSU); //turn the image only black or while

return image;

}

This will edit the image to only show a black background and white dice with black spots. After this we use the find contours function to detect all squares within the edited image. We first check all those squares if it’s the correct size, and if they are we can follow up by checking the dots within the approved squares. We then crop an area of the approved squares and analyze those points only within that cropped area. With the cropped area, we perform the following algorithm:

int Application::countPips(cv::Mat dice)

{

// resize

cv::resize(dice, dice, cv::Size(150, 150));

dice = changeImage(dice);

// floodfill

cv::floodFill(dice, cv::Point(0, 0), cv::Scalar(255));

cv::floodFill(dice, cv::Point(0, 149), cv::Scalar(255));

cv::floodFill(dice, cv::Point(149, 0), cv::Scalar(255));

cv::floodFill(dice, cv::Point(149, 149), cv::Scalar(255));

// search for blobs

cv::SimpleBlobDetector::Params params;

// filter by interia defines how elongated a shape is.

params.filterByInertia = true;

params.minInertiaRatio = 0.5;

// will hold our keyponts

std::vector<cv::KeyPoint> keypoints;

// create new blob detector with our parameters

cv::Ptr<cv::SimpleBlobDetector> blobDetector = cv::SimpleBlobDetector::create(params);

// detect blobs

blobDetector->detect(dice, keypoints);

// return number of pips

return (int)keypoints.size();

}

This algorithm will return the value of the dice it has analyzed. With this done, it will loop through all the approved squares and once all have been approved, we can draw the results on screen. And to show to process, we use the following function to display the webcam picture:

cv::imshow("Dice Image", unprocessFrame);

# Tips using the Engine

* Please use the Application file base when starting a new project.
* Don’t be afraid of using components and creating new components for fringe cases.
* Handle your buttons in the Handle Events function
* Events -> Update -> Render
* Redner order matters, double check that first if your entity is not drawing itself on screen.

# Plans regarding improving the Engine

In the future of this engine, I wish to add a state system that can check game states wisely, for example, moving from a menu to the game to an inventory and so on.

Next, I wish to add a working physics engine within, but this can become tricky if I wish to stay within the boundaries of SDL. Emscripten functionality would also be a fun test.

There are still a ton of bugs within the system, but nothing generally game breaking.

# Credits and references

Thank you, Henri, for giving me this project, all my friends and family for cheering me on and Stack Overflow for a lot of C++ help. Also, I’d like to thank the following links:

<https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx>  
For the general tutorial of building an engine using SDL

<https://github.com/hashstash/EventManager-Cpp> for the Event Manager system

<https://youtu.be/Qn2D6lq08gk?list=LLSpwLPlrBCMZMqdewAKIa_Q> For the help with TMX file loading