SDL2.0 Game Engine Core

1.0

Generated by Doxygen 1.8.2

Fri Mar 11 2016 08:14:24

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 CAnimation Class Reference

A SDL Animation class.

```
#include <CAnimation.h>
```

Public Member Functions

- CAnimation ()
- void OnAnimate ()

  An animation loop.
- void SetFrameRate (int Rate)
- void SetCurrentFrame (int Frame)
- int GetCurrentFrame ()

Public Attributes

- int MaxFrames
- bool Oscillate

4.1.1 Detailed Description

A SDL Animation class.

We'll be creating a new class to handle Animation, and in the next tutorial we will create a class to handle Entities. Please keep in mind that these two things are seperate, and while I know they could be one class, I don't wish to take that approach. So please hold back your criticism.

Some explanation on what this class is all about now. There is one basic element of animation that we need to handle, that is the current frame of the animation. Take the Yoshi image below for example (we'll be using him in this tutorial). You'll notice we have 8 frames of Yoshi on one image. Each frame would then be labeled 0, 1, 2 from Top to Bottom.

Remember the second tutorial where we create a function to draw part of an image? Well, if we take that function in conjunction with the frame of the animation, voila!

Definition at line 77 of file CAnimation.h.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 CAnimation::CAnimation ( )

Definition at line 3 of file CAnimation.cpp.

4.1.3 Member Function Documentation

4.1.3.1 int CAnimation::GetCurrentFrame ( )

Definition at line 60 of file CAnimation.cpp.

4.1.3.2 void CAnimation::OnAnimate ( )

An animation loop.

By taking the Old Time in milliseconds plus the desired frame rate, we can check it against how long the SDL has been running. So, for example, we just started our program. SDL_GetTicks is 0, and OldTime is 0. Our desired frame rate is 1 frame per second. So FrameRate = 1000 (milliseconds). So, is 0 + 1000 greater than 0? Yes, thus we will skip over the function and wait. But once 0 + 1000 is less than SDL_GetTicks, that must mean 1 second has passed. So we increment the frame, and then reset OldTime to the current time, and start over.

We already know what the OldTime and such do, but what about the rest? For now, look at the else statement of the Oscillate if statement. You'll notice we're simply checking if the CurrentFrame has exceeded the Max Frames. If it has, reset back to 0. Pretty simple. Then below that, outside of the block, we increase to the next frame.

Now, the more confusing part is the Oscillate if statement. This is where the FrameInc variable comes in. Basically, the FrameInc is set to 1 or -1, depending on how we are increasing or decreasing the frames. Remember, Oscillating causes the frames to go from 0 to 9 back to 0. So if the FrameInc is greater than 0, we are increasing the frames, otherwise we are decreasing frames. The innermost if statements basically inverse the FrameInc if we reached the end of 0, or MaxFrames.

Definition at line 22 of file CAnimation.cpp.

4.1.3.3 void CAnimation::SetCurrentFrame (int *Frame*)

Definition at line 54 of file CAnimation.cpp.

4.1.3.4 void CAnimation::SetFrameRate (int *Rate*)

Definition at line 50 of file CAnimation.cpp.

4.1.4 Member Data Documentation

4.1.4.1 int CAnimation::MaxFrames

Definition at line 89 of file CAnimation.h.

4.1.4.2 bool CAnimation::Oscillate

Definition at line 91 of file CAnimation.h.

The documentation for this class was generated from the following files:

- Sokoban/CAnimation.h
- Sokoban/CAnimation.cpp

4.2 CApp Class Reference

A core program class.

```
#include <CApp.h>
```

Inheritance diagram for CApp:

Collaboration diagram for CApp:

Public Member Functions

- CApp ()

  A constructor.
- int OnExecute ()

  A game loop.
- bool OnInit ()

  An game initialization.
- void OnEvent (SDL_Event ∗Event)

  An event procesing delegate.
- void OnLButtonDown (int mX, int mY)

  Mouse event.
- void OnKeyDown (SDL_Keysym sym, Uint16 mod, Uint16 unicode)
- void OnKeyUp (SDL_Keysym sym, Uint16 mod, Uint16 unicode)
- void OnExit ()

  The OnExit function handles the SDL_QUIT events. Now that we have the prototype, lets define what it does.
- void OnLoop ()
- void OnRender ()
- void OnCleanup ()

  An game destructor.
- void Reset ()

  Reset the effects.
- void SetCell (int ID, int Type)

  Set an effect.

4.2.1 Detailed Description

A core program class.

A more elaborate class description.

Definition at line 96 of file CApp.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 CApp::CApp ( )

A constructor.

A more elaborate description of the constructor in H.

A more elaborate description of the constructor in CPP. < initialize the surfaces to NULL.

Definition at line 31 of file CApp.cpp.

4.2.3 Member Function Documentation

4.2.3.1 void CApp::OnCleanup ( )

An game destructor.

We basically quit out of SDL. You should take note that in this function is where you would free other surfaces as well. This keeps all your code centralized to the function its performing. $<$ Free the surfaces.

$<$ To keep things tidy, lets also set the Surf_Display pointer to NULL.

we are basically encapsulating the basic functions of a game within the Entity class? We have to call those functions now in the respective CApp functions.

We are basically running through each Entity in our vector, and calling the OnLoop function. Simple enough! (And we're doing an error checking so we don't call any NULL pointers).

$<$ Close AUDIO.

$<$ Quit SDL.

Definition at line 7 of file CApp_OnCleanup.cpp.

4.2.3.2 void CApp::OnEvent (SDL_Event ∗ *Event*) [virtual]

An event procesing delegate.

Calling CEvent object to resolve the SDL_Event. We have the event class setup, now lets actually link events to our new class structure.

Reimplemented from CEvent.

Definition at line 14 of file CApp_OnEvent.cpp.

4.2.3.3 int CApp::OnExecute ( )

A game loop.

You'll notice some new variables, but let's look at what is happening first. First, we try to Initialize our game, if it fails we return -1 (an error code), thus closing our program. If everything is good, we continue on to the game loop. Within the game loop we use SDL_PollEvent to check for events, and pass them one at a time to OnEvent. Once done with Events, we go to OnLoop for move data around and what not, and then render our game. We repeat this indefinitly. If the user exits the game, we proceed to OnCleanup cleaning up any resources. Simple enough.

Now, lets look at SDL_Event and SDL_PollEvent. The first is a structure that holds information about events. The second is a function that will grab any events waiting in the queue. This queue can have any number of events, which is the reason why we have to loop through them. So, for example, lets say the user presses A and moves the mouse during the OnRender() function. SDL will detect this and put two events in the queue, one for a key press and one for a mouse move. We can grab this event from the queue by using the SDL_PollEvent, and then passing it to OnEvent to handle it accordingly. Once there are no more events in the queue, SDL_PollEvent will return false, thus exiting out of the Event queue loop.

The other variable added, Running, is our own. This is our exit out of the game loop. When this is set to false, it will end the program, and in turn exit the program. So, for example, if the user presses the Escape key we can set this variable to false, quitting the game. $<$ The first is a structure that holds information about events.

$<$ Get back

Definition at line 54 of file CApp.cpp.

4.2.3.4 void CApp::OnExit ( ) [virtual]

The OnExit function handles the SDL_QUIT events. Now that we have the prototype, lets define what it does.

The type above we are looking for is the request to close the window (i.e., when the user clicks the X button). If that event happens to take place, we set Running to false, thus ending our program. Simple enough.

Reimplemented from CEvent.

Definition at line 101 of file CApp_OnEvent.cpp.

4.2.3.5 bool CApp::OnInit ( )

An game initialization.

The first thing we need to do is start up SDL itself, so we can access its functions. We are telling SDL to Initialize everything it has; there are other parameters you can pass, but understanding them at this point is not important. The next function we use is SDL_SetVideoMode. This bad boy is what creates our window, and our surface. It takes 4 parameters: The width of the window, the height of the window, the bit resolution of the window (recommended to be 16 or 32), and then display flags. There are quite a few display flags, but the ones shown above are fine for now. The first flag tells SDL to use hardware memory for storing our images and such, and the second flag tells SDL to use double buffering (which is important if you don't want flickering on your screen). Another flag that may interest you now is SDL_FULLSCREEN, which makes the window go fullscreen. load an image.

$<$ Now, lets set the MaxFrames.

$<$ If you want to see your animation Oscillate.

Definition at line 7 of file CApp_OnInit.cpp.

4.2.3.6 void CApp::OnKeyDown (SDL_Keysym *sym,* Uint16 *mod,* Uint16 *unicode*) [virtual]

Reimplemented from CEvent.

Definition at line 56 of file CApp_OnEvent.cpp.

4.2.3.7 void CApp::OnKeyUp (SDL_Keysym *sym,* Uint16 *mod,* Uint16 *unicode*) [virtual]

Reimplemented from CEvent.

Definition at line 80 of file CApp_OnEvent.cpp.

4.2.3.8 void CApp::OnLButtonDown (int *mX,* int *mY*) [virtual]

Mouse event.

First, we are doing the reverse of what we did with translating to X and Y from an ID, this time we are translating to an ID. We then make sure that that cell hasn't already been taken, if it has, we return out of the function. Next, we are checking which players turn it is, set the cell appropriately, and then switch turns.

Reimplemented from CEvent.

Definition at line 22 of file CApp_OnEvent.cpp.

4.2.3.9 void CApp::OnLoop ( )

$<$ now to make our animation loop.

we are basically encapsulating the basic functions of a game within the Entity class? We have to call those functions now in the respective CApp functions.

We are basically running through each Entity in our vector, and calling the OnLoop function. Simple enough! (And we're doing an error checking so we don't call any NULL pointers).

Definition at line 10 of file CApp_OnLoop.cpp.

4.2.3.10 void CApp::OnRender ( )

< You should notice that your image is drawn at 100, 100 and only part of it is being displayed.

< lets actually draw the image.

< to make it actually animate.

we are basically encapsulating the basic functions of a game within the Entity class? We have to call those functions now in the respective CApp functions.

We are basically running through each Entity in our vector, and calling the OnLoop function. Simple enough! (And we're doing an error checking so we don't call any NULL pointers).

Notice a new function here SDL_Flip. This basically refreshes the buffer and displays Surf_Display onto the screen. This is called double buffering. It's the process of drawing everything into memory, and then finally drawing everything to the screen. If we didn't do this, we would have images flickering on the screen. Remember the SDL_DOUBLEBUF flag? This is what turns double buffering on.

Definition at line 3 of file CApp_OnRender.cpp.

4.2.3.11 void CApp::Reset ( )

Reset the effects.

This loop will set every cell in the grid to GRID_TYPE_NONE, meaning that all the cells are empty. We're going to have to do this at the very beginning when our program is loaded.

Definition at line 80 of file CApp.cpp.

4.2.3.12 void CApp::SetCell (int *ID,* int *Type*)

Set an effect.

This function takes two arguments, the first is the cell ID to change, and the second is the type to change it to. We have to conditions here, the first is to make sure we don't go outside the bounds of the array (if we did it would likely crash our program), and the second is to make sure we passed an appropriate type. Simple enough.

Definition at line 90 of file CApp.cpp.

The documentation for this class was generated from the following files:

- Sokoban/CApp.h
- Sokoban/CApp.cpp
- Sokoban/CApp_OnCleanup.cpp
- Sokoban/CApp_OnEvent.cpp
- Sokoban/CApp_OnInit.cpp
- Sokoban/CApp_OnLoop.cpp
- Sokoban/CApp_OnRender.cpp

4.3 CArea Class Reference

```
#include <CArea.h>
```

Collaboration diagram for CArea:

Public Member Functions

- CArea ()
  A constructor.

- bool OnLoad (char ∗File, SDL_Surface ∗Screen_Display)

  Then we have our Load function.
- void OnRender (SDL_Surface ∗Surf_Display, int CameraX, int CameraY)

  Next, we have the Render function.
- void OnCleanup ()

  Lastly, we have our cleanup function that frees the surface and clears the maps.
- CMap ∗ GetMap (int X, int Y)
- CTile ∗ GetTile (int X, int Y)

Public Attributes

- std::vector< CMap > MapList

Static Public Attributes

- static CArea AreaControl

  we declare the static object.

4.3.1 Detailed Description

Definition at line 47 of file CArea.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 CArea::CArea ( )

A constructor.

and then set the AreaSize to 0.

Definition at line 18 of file CArea.cpp.

4.3.3 Member Function Documentation

4.3.3.1 CMap ∗ CArea::GetMap (int *X*, int *Y*)

Definition at line 116 of file CArea.cpp.

4.3.3.2 CTile ∗ CArea::GetTile (int *X*, int *Y*)

Definition at line 130 of file CArea.cpp.

4.3.3.3 void CArea::OnCleanup ( )

Lastly, we have our cleanup function that frees the surface and clears the maps.

Details.

Definition at line 108 of file CArea.cpp.

4.3.3.4 bool CArea::OnLoad (char ∗ *File,* SDL_Surface ∗ *Screen_Display*)

Then we have our Load function.

It works just like the Load function within CMap, except there are a few differences. We have to load tileset first. We try to load this tileset into the surface, and if it fails return false. We then load the size of the Area. Okay, after that we have two loops, just like maps, that will go through and grab each map filename. It will then create a map object, load the map, set the tileset to be used, and then push it into the list. Pretty simply and straight forward.

Definition at line 26 of file CArea.cpp.

4.3.3.5 void CArea::OnRender (SDL_Surface ∗ *Surf_Display,* int *CameraX,* int *CameraY*)

Next, we have the Render function.

We calculate the actual Map width and height, in pixels, first. This will let us find the first map to render. Some explanation first of what I am trying to do. Since an area can be of any size, like 100x100 maps, we don't want to go through all the trouble and render every single map. We only want to render the maps that are visible in the screen area. For our size of a screen, 640x480, only a possible 4 maps can be visible at one time (like standing at the corner of 4 maps). So what we have to do is calculate the first Map ID to grab. This is going to the first map to render. From the first ID, we know the next 3 maps to render. The one to the right of the first map, the one at the bottom of the first map, and the one at the bottom right of the first map.

We have a guy, the circle, in the very center of the screen, represented by the red square. The other squares are each Map, and we have a Camera position of -700, -700. Why negative? Think about it, the screen itself never really moves, everything else does. So, for something to move up, it must increase its Y in the negative direction. Same with the X coordinate. So, to get to the 4th map, we have to move the area in the negative direction. Now, notice the grayed out maps, these are maps not visible within the players view, so they shouldn't be rendered. To figure out the First ID, which is 4 in this case, we need to use the specified Camera coordinates. We translate those Camera coordinates into Map coordinates. So, taking -(-700) / 640 (which is 40 ∗ 16, remember the MapWidth calculation done above) we get 1 (dropping the decimal because its an integer operation). This is the X coordinate in Maps, but we aren't done. We then calculate the Y coordinate in Maps the same way -(-700) / 640, but we multiply that by the AreaSize. That's because we are grabbing the ID. So, it would become 1 ∗ 3, which is 3, and adding that to the first calculation of 1, we get 4. And what do you know, the First ID on the map!

Okay, so we go through each of the four maps now. Since I did a regular loop of i < 4, we need to figure out how to add that to the First ID, to actually figure out each of the four Map IDs. This is done by taking the First ID as an offset first. We then take i, our position in the loop, divide by two and multiply by the area size. What does this do? Much like the Map class, it creates a pattern, 0, 0, 1, 1. Same with the last operation i % 2, it creates a pattern of 0, 1, 0, 1. This then gives us the correct pattern of, 4, 5, 7, 8. Which are the correct map to render.

We do a little check to make sure the ID is good, since the ID may not exist. And now the last calculation (yes, I know, a lot of complex calculations). It works just like the way we calculated which tile to grab on the tileset. It's turning an ID into actual pixel coordinates, and then offsetting those coordinates by the camera (making it seem like it moved). We then finally render the map, passing the coordinates of where to draw it.

Definition at line 85 of file CArea.cpp.

4.3.4 Member Data Documentation

4.3.4.1 CArea CArea::AreaControl [static]

we declare the static object.

Definition at line 49 of file CArea.h.

4.3.4.2 std::vector<CMap> CArea::MapList

Definition at line 52 of file CArea.h.

The documentation for this class was generated from the following files:

- Sokoban/CArea.h
- Sokoban/CArea.cpp

4.4 CCamera Class Reference

```
#include <CCamera.h>
```

Collaboration diagram for CCamera:

Public Member Functions

- CCamera ()
- void OnMove (int MoveX, int MoveY)
- int GetX ()
- int GetY ()
- void SetPos (int X, int Y)
- void SetTarget (float ∗X, float ∗Y)

Public Attributes

- int TargetMode

Static Public Attributes

- static CCamera CameraControl

4.4.1 Detailed Description

Definition at line 23 of file CCamera.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 CCamera::CCamera ( )

Definition at line 13 of file CCamera.cpp.

4.4.3 Member Function Documentation

4.4.3.1 int CCamera::GetX ( )

Definition at line 26 of file CCamera.cpp.

4.4.3.2 int CCamera::GetY ( )

Definition at line 38 of file CCamera.cpp.

4.4.3.3 void CCamera::OnMove (int *MoveX,* int *MoveY*)

Definition at line 21 of file CCamera.cpp.

4.4.3.4 void CCamera::SetPos (int *X,* int *Y*)

Definition at line 50 of file CCamera.cpp.

4.4.3.5 void CCamera::SetTarget (float ∗ *X,* float ∗ *Y*)

Definition at line 55 of file CCamera.cpp.

4.4.4 Member Data Documentation

4.4.4.1 CCamera CCamera::CameraControl `[static]`

Definition at line 25 of file CCamera.h.

4.4.4.2 int CCamera::TargetMode

Definition at line 35 of file CCamera.h.

The documentation for this class was generated from the following files:

- Sokoban/CCamera.h
- Sokoban/CCamera.cpp

4.5 CEntity Class Reference

A entity class.

`#include <CEntity.h>`

Inheritance diagram for CEntity:

Collaboration diagram for CEntity:

Public Member Functions

- CEntity ()

    A constructor.
- virtual ∼CEntity ()
- virtual bool OnLoad (char ∗File, int Width, int Height, int MaxFrames, SDL_Surface ∗Screen_Display)
- virtual void OnLoop ()

    On Loop.
- virtual void OnRender (SDL_Surface ∗Surf_Display)

    On Render.
- virtual void OnCleanup ()
- virtual void OnAnimate ()

    On Animate.
- virtual bool OnCollision (CEntity ∗Entity)

    On Collision.

- void OnMove (float MoveX, float MoveY)

    On Move.
- void StopMove ()

    Stop Move.
- bool Jump ()
- bool Collides (int oX, int oY, int oW, int oH)

    Collides.

Public Attributes

- float X
- float Y
- int Width
- int Height
- int AnimState
- bool MoveLeft
- bool MoveRight
- int Type
- bool Dead
- int Flags
- float MaxSpeedX
- float MaxSpeedY

Static Public Attributes

- static std::vector< CEntity ∗ > EntityList

Protected Attributes

- CAnimation Anim_Control
- SDL_Surface ∗ Surf_Entity
- bool CanJump
- float SpeedX
- float SpeedY
- float AccelX
- float AccelY
- int CurrentFrameCol
- int CurrentFrameRow
- int Col_X
- int Col_Y
- int Col_Width
- int Col_Height

4.5.1 Detailed Description

A entity class.

Entities, for all gaming purposes, are anything that can be interacted with in any way, shape, or form. Some examples might be a monster or a treasure chest that you can open. In this sense, practically everything within the game that moves is an Entity. A rock that is part of a map, which never moves, is not an entity.

Definition at line 156 of file CEntity.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 CEntity::CEntity ( )

A constructor.

Everything is simply set to 0, NULL, or some respective value. One thing to notice is the MaxSpeed variables. Set these to whatever you like. Usually this will depend on the particular entity that is in the game. Such as Mario will be able to move faster than a Goomba. Notice that I have grouped two classes into one file; that is okay to do sometimes! < These are used to determine certain properties about an entity. For example, an entity that is affected by gravity will have the ENTITY_FLAG_GRAVITY turned on. An Entity that can go through walls, will have ENTITY_FLAG_GHOST turned on. An entity that only collides with the map, and not other entities, will have ENTITY_FLAG_MAPONLY turned on. Now, notice how we have them numbered in hex. This is because we're treating the Flags in binary. This way, we can activate multiple flags at once:

Definition at line 16 of file CEntity.cpp.

4.5.2.2 CEntity::∼CEntity ( ) [virtual]

Definition at line 57 of file CEntity.cpp.

4.5.3 Member Function Documentation

4.5.3.1 bool CEntity::Collides (int *oX,* int *oY,* int *oW,* int *oH*)

Collides.

Next, we have a function that determines collision of two boxes. I will say right now that this function was borrowed from Cone3D (), and then modified. If it works, why fix it?

Anyway, lets take a look at it to see if I can help explain. At the top we define 8 variables, to define 8 sides. 4 sides are for the first object, and 4 sides are for the second object. Basically, we are doing bounding box collision:

So, in a very basic sense, we want to check if two boxes have overlapped. The easiest method, is to determine if we have not collided on all 4 sides. So first, we determine the locations of all the sides:

The top and left, are the X,Y of the object, and the right and bottom, are the X+Width-1, Y+Height-1. We subtract one in order to get the true coordinate of the side.

Next, we finally do the checking. What we do here is to check if one of the 4 sides of the first object are out of range of the second object. If one is, then we cannot possibly be colliding. But, if all 4 checks fail, we are colliding. So, if bottom1 is less than top2, that means the Bottom side of the first object is higher up on the screen then the Top of the second object:

Finally, if all checks pass, then we return true for a collision. One note, remember that the Col_ variables are used to simply offset the size of the entity. If the entity is 32 width, but we offset by 2, that means when we check collision we use the Width of 30.

Definition at line 277 of file CEntity.cpp.

4.5.3.2 bool CEntity::Jump ( )

Definition at line 255 of file CEntity.cpp.

4.5.3.3 void CEntity::OnAnimate ( ) [virtual]

On Animate.

This is fairly straight forward. Like I said above, depending on which direction we are moving, we set the animation.

Reimplemented in CPlayer.

Definition at line 136 of file CEntity.cpp.

4.5.3.4 void CEntity::OnCleanup ( ) [virtual]

Reimplemented in CPlayer.

Definition at line 124 of file CEntity.cpp.

4.5.3.5 bool CEntity::OnCollision (CEntity * *Entity*) [virtual]

On Collision.

Yay, an empty function. This function will be called whenever two entities hit each other. This entire time we've been creating an class for basic entities. In order to take advantage of this function, you will need to create our own class (like CPlayer, which we'll do later on in this tutorial) and override this function. This is because the action taken by an entity depends upon it's type. Like my earlier examples, a Goomba will not react the same way to a collision with a spike as Mario would.

Reimplemented in CPlayer.

Definition at line 153 of file CEntity.cpp.

4.5.3.6 bool CEntity::OnLoad (char * *File,* int *Width,* int *Height,* int *MaxFrames,* SDL_Surface * *Screen_Display*) [virtual]

Reimplemented in CPlayer.

Definition at line 60 of file CEntity.cpp.

4.5.3.7 void CEntity::OnLoop ( ) [virtual]

On Loop.

Lots of new things here. First, remember MoveLeft and MoveRight are used to indicate if the entity is moving left or right (duh). If both are these are false, we call StopMove, which will handle settings Acceleration in the reverse direction so we come to a gradual stop. Below that, if MoveLeft or MoveRight are true, we give AccelX a value. 0.5 is an arbitrary number, you probably want to mess with this until you find a value that you like. Just below that, we have a condition checking if gravity is turned on for that particular entity. If it is, we set AccelY. This will cause our entity to fall down. Again, this is whatever value you want. The larger the number, the faster you will fall. Next, below that we have our SpeedX and SpeedY. Again, we increase the Speed of the entity over time in correlation to our Speed Factor. I want my player to increase in speed 1 pixel per second. This would be:

Reimplemented in CPlayer.

Definition at line 79 of file CEntity.cpp.

4.5.3.8 void CEntity::OnMove (float *MoveX,* float *MoveY*)

On Move.

This function is quite a bit to take in. Lets see if we can straighten it all out. First, MoveX and MoveY are the where we want to move our entity. Remember up above, we call this passing our SpeedX and SpeedY. So, calling OnMove(1, 0) we would want to move 1 pixel to the right every second. At the top, we check our Speed, and if 0, we don't do anything (duh). Below that, we do something I explained earlier. We multiply MoveX and MoveY by the speed factor. This gives us the correct movement per second. Just below that, we figure out the values for two new variables, NewX and NewY. These will be our X and Y increments. Remember, I mentioned above that this method is pixel perfect, so we need to inch closer and closer to our desired position. Now, we set our NewX and NewY to our speed factor because this is how much we should use to inch closer and closer to our desired position.

Below that, we have a seemingly infinite loop. This is where the movement magic happens and collision. First, we have a condition to check if the entity is a ghost, if it is, we increase the X and Y regardless if a collision has happened. We still call PosValid though, because this will notify entities of any collision that have taken place. Notice in PosValid we pass the current X and Y, along with adding NewX and NewY. This is where we want to go, so we check if that spot is empty.

If the entity is not a ghost, we do normal collision. We check X and Y separatly because we may be able to move up but not forward (think jumping into a wall). If the position is valid (empty), we increase the X or Y, if not, we set SpeedX or SpeedY to 0 which causes you to immediately stop (this is as close to reality as we need, if you do run into a wall you usually stop immediately). Now, below that, we change MoveX and MoveY depending upon our NewX. Say our MoveX was 2, and our NewX was 0.5. We would slowly decrease MoveX until it reaches 0. Once it does, we have come to our desired position and we are done. The 8 if statements below check if MoveX and MoveY have reached 0, and break out of the loop. Notice we multiply the AccelX by the Speed Factor. That's because we want to increase the speed 1 pixel every second. If I wanted to increase the speed 1 pixel every loop, I would leave the speed factor out of it. Now, when OnMove receives these Speed variables, it also has to figure in the speed factor:

(MoveX and MoveY are the arguments passed to OnMove). Now, the SpeedX and SpeedY only determine how fast we want to move per second. So we have to multiply it by the speed factor. A little hint for you people out there, the entire speed of the game is determined by the Speed Factor! If we changed the Speed Factor of the game, it would determine how fast anything in the games moves. Slow motion deaths anyone?

Now, whenever an Entity moves, there are some things to consider. Say in one loop we are supposed to move the Entity 4 pixels (slow computer perhaps, or really fast entity). Now, lets say 2 pixels away there is a wall. Now, if we checked 4 pixels in front of use, we wouldn't let our guy move. But, we still want him to move as close as he can. So, to move this 2 pixel distance, we have to check every pixel along the way. Now, you might be thinking this will be slow. Well, not really. Someone with a 100 FPS who wants to move 5 pixels a second (a reasonable speed), will move less than one pixel every loop (0.05 to be exact). So, every loop, this entity moves 0.05 pixels over. That means we have 1 or 2 checks when checking for collisions for that entity. More explanation on this later when we actually implement moving.

Definition at line 165 of file CEntity.cpp.

4.5.3.9 void CEntity::OnRender (SDL_Surface * *Surf_Display*) `[virtual]`

On Render.

First, we have added the Camera X and Y to the rendering coordinates for the entity. This makes the entities render in connection to where the camera is. This way, if the camera moves, all the entities will in respect to the camera position. Next, we have added CurrentFrameCol and CurrentFrameRow to the X2 and Y2 of OnDraw. This makes it so we can set different animations.

The first column has Yoshi facing the left, while the second column has Yoshi facing to the right. When we move to the left, we will want to set CurrentFrameCol = 0, so that Yoshi appears to be walking left. And when we move to the right, we will set CurrentFrameCol = 1, so that Yoshi appears to be walking to the right.

Reimplemented in CPlayer.

Definition at line 118 of file CEntity.cpp.

4.5.3.10 void CEntity::StopMove ( )

Stop Move.

This one is fairly straight forward. Depending on our Speed, we set the Acceleration in the reverse direction. So, if we are moving right, we set the acceleration to the left. The last if statement is simply a boundary to how slow we need to be going until we stop. If we didn't do that, it's possible that we could be stuck in a loop of moving left to right.

Definition at line 240 of file CEntity.cpp.

4.5.4 Member Data Documentation

4.5.4.1 float CEntity::AccelX [protected]

Definition at line 191 of file CEntity.h.

4.5.4.2 float CEntity::AccelY [protected]

Definition at line 192 of file CEntity.h.

4.5.4.3 CAnimation CEntity::Anim_Control [protected]

Definition at line 161 of file CEntity.h.

4.5.4.4 int CEntity::AnimState

Definition at line 173 of file CEntity.h.

4.5.4.5 bool CEntity::CanJump [protected]

Definition at line 179 of file CEntity.h.

4.5.4.6 int CEntity::Col_Height [protected]

Definition at line 206 of file CEntity.h.

4.5.4.7 int CEntity::Col_Width [protected]

Definition at line 205 of file CEntity.h.

4.5.4.8 int CEntity::Col_X [protected]

Definition at line 203 of file CEntity.h.

4.5.4.9 int CEntity::Col_Y [protected]

Definition at line 204 of file CEntity.h.

4.5.4.10 int CEntity::CurrentFrameCol [protected]

Definition at line 199 of file CEntity.h.

4.5.4.11 int CEntity::CurrentFrameRow [protected]

Definition at line 200 of file CEntity.h.

4.5.4.12 bool CEntity::Dead

Definition at line 184 of file CEntity.h.

4.5.4.13 std::vector< CEntity * > CEntity::EntityList [static]

Definition at line 158 of file CEntity.h.

4.5.4.14 int CEntity::Flags

Definition at line 185 of file CEntity.h.

4.5.4.15 int CEntity::Height

Definition at line 170 of file CEntity.h.

4.5.4.16 float CEntity::MaxSpeedX

Definition at line 195 of file CEntity.h.

4.5.4.17 float CEntity::MaxSpeedY

Definition at line 196 of file CEntity.h.

4.5.4.18 bool CEntity::MoveLeft

Definition at line 175 of file CEntity.h.

4.5.4.19 bool CEntity::MoveRight

Definition at line 176 of file CEntity.h.

4.5.4.20 float CEntity::SpeedX [protected]

Definition at line 188 of file CEntity.h.

4.5.4.21 float CEntity::SpeedY [protected]

Definition at line 189 of file CEntity.h.

4.5.4.22 SDL_Surface* CEntity::Surf_Entity [protected]

Definition at line 163 of file CEntity.h.

4.5.4.23 int CEntity::Type

Definition at line 182 of file CEntity.h.

4.5.4.24 int CEntity::Width

Definition at line 169 of file CEntity.h.

4.5.4.25 float CEntity::X

Definition at line 166 of file CEntity.h.

4.5.4.26 float CEntity::Y

Definition at line 167 of file CEntity.h.

The documentation for this class was generated from the following files:

- Sokoban/CEntity.h
- Sokoban/CEntity.cpp

4.6 CEntityCol Class Reference

`#include <CEntity.h>`

Collaboration diagram for CEntityCol:

Public Member Functions

- CEntityCol ()

  A constructor.

Public Attributes

- CEntity ∗ EntityA
- CEntity ∗ EntityB

Static Public Attributes

- static std::vector< CEntityCol > EntityColList

4.6.1 Detailed Description

Definition at line 245 of file CEntity.h.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 CEntityCol::CEntityCol ( )

A constructor.

Everything is simply set to 0, NULL, or some respective value.

Definition at line 9 of file CEntityCol.cpp.

4.6.3 Member Data Documentation

4.6.3.1 CEntity∗ CEntityCol::EntityA

Definition at line 250 of file CEntity.h.

4.6.3.2 CEntity∗ CEntityCol::EntityB

Definition at line 251 of file CEntity.h.

4.6.3.3 std::vector< CEntityCol > CEntityCol::EntityColList [static]

Definition at line 247 of file CEntity.h.

The documentation for this class was generated from the following files:

- Sokoban/CEntity.h
- Sokoban/CEntityCol.cpp

4.7 CEvent Class Reference

An event class.

`#include <CEvent.h>`

Inheritance diagram for CEvent:

Public Member Functions

- CEvent ()
- virtual ∼CEvent ()
- virtual void OnEvent (SDL_Event ∗Event)

    An event procesor.
- virtual void OnInputFocus ()
- virtual void OnInputBlur ()
- virtual void OnKeyDown (SDL_Keysym sym, Uint16 mod, Uint16 unicode)
- virtual void OnKeyUp (SDL_Keysym sym, Uint16 mod, Uint16 unicode)
- virtual void OnMouseFocus ()
- virtual void OnMouseBlur ()
- virtual void OnMouseMove (int mX, int mY, int relX, int relY, bool Left, bool Right, bool Middle)
- virtual void OnMouseWheel (bool Up, bool Down)
- virtual void OnLButtonDown (int mX, int mY)
- virtual void OnLButtonUp (int mX, int mY)
- virtual void OnRButtonDown (int mX, int mY)
- virtual void OnRButtonUp (int mX, int mY)
- virtual void OnMButtonDown (int mX, int mY)
- virtual void OnMButtonUp (int mX, int mY)
- virtual void OnJoyAxis (Uint8 which, Uint8 axis, Sint16 value)
- virtual void OnJoyButtonDown (Uint8 which, Uint8 button)
- virtual void OnJoyButtonUp (Uint8 which, Uint8 button)
- virtual void OnJoyHat (Uint8 which, Uint8 hat, Uint8 value)
- virtual void OnJoyBall (Uint8 which, Uint8 ball, Sint16 xrel, Sint16 yrel)
- virtual void OnMinimize ()
- virtual void OnRestore ()
- virtual void OnResize (int w, int h)
- virtual void OnExpose ()
- virtual void OnExit ()
- virtual void OnUser (Uint8 type, int code, void ∗data1, void ∗data2)

4.7.1 Detailed Description

An event class.

Alongside the basics of game development is something called Events. All videogames, from pong to the highly complex PC games and console titles, use events to interact with the player. These events can come from keyboards, mice, joysticks, gamepads, and so on, or events from our operating system. It's important to understand how events work if we are to appropriately interact a user with a game. We've already been using events, but only for closing our window, now we'll look at how to receive events from the user.

Definition at line 23 of file CEvent.h.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 CEvent::CEvent ( )

Definition at line 27 of file CEvent.cpp.

4.7.2.2 CEvent::∼CEvent ( ) `[virtual]`

Definition at line 30 of file CEvent.cpp.

4.7.3 Member Function Documentation

4.7.3.1 void CEvent::OnEvent (SDL_Event ∗ *Event*) `[virtual]`

An event procesor.

Lots of code, but all the SDL events should be covered. What we basically are doing is taking an SDL_Event pointer, and switching through the types, and then calling the appropriate function. It just looks like a lot since they are quite a bit of events.

Reimplemented in CApp.

Definition at line 38 of file CEvent.cpp.

4.7.3.2 void CEvent::OnExit ( ) `[virtual]`

Reimplemented in CApp.

Definition at line 264 of file CEvent.cpp.

4.7.3.3 void CEvent::OnExpose ( ) `[virtual]`

Definition at line 260 of file CEvent.cpp.

4.7.3.4 void CEvent::OnInputBlur ( ) `[virtual]`

Definition at line 176 of file CEvent.cpp.

4.7.3.5 void CEvent::OnInputFocus ( ) `[virtual]`

Definition at line 172 of file CEvent.cpp.

4.7.3.6 void CEvent::OnJoyAxis (Uint8 *which,* Uint8 *axis,* Sint16 *value*) [virtual]

Definition at line 228 of file CEvent.cpp.

4.7.3.7 void CEvent::OnJoyBall (Uint8 *which,* Uint8 *ball,* Sint16 *xrel,* Sint16 *yrel*) [virtual]

Definition at line 244 of file CEvent.cpp.

4.7.3.8 void CEvent::OnJoyButtonDown (Uint8 *which,* Uint8 *button*) [virtual]

Definition at line 232 of file CEvent.cpp.

4.7.3.9 void CEvent::OnJoyButtonUp (Uint8 *which,* Uint8 *button*) [virtual]

Definition at line 236 of file CEvent.cpp.

4.7.3.10 void CEvent::OnJoyHat (Uint8 *which,* Uint8 *hat,* Uint8 *value*) [virtual]

Definition at line 240 of file CEvent.cpp.

4.7.3.11 void CEvent::OnKeyDown (SDL_Keysym *sym,* Uint16 *mod,* Uint16 *unicode*) [virtual]

Reimplemented in CApp.

Definition at line 180 of file CEvent.cpp.

4.7.3.12 void CEvent::OnKeyUp (SDL_Keysym *sym,* Uint16 *mod,* Uint16 *unicode*) [virtual]

Reimplemented in CApp.

Definition at line 184 of file CEvent.cpp.

4.7.3.13 void CEvent::OnLButtonDown (int *mX,* int *mY*) [virtual]

Reimplemented in CApp.

Definition at line 204 of file CEvent.cpp.

4.7.3.14 void CEvent::OnLButtonUp (int *mX,* int *mY*) [virtual]

Definition at line 208 of file CEvent.cpp.

4.7.3.15 void CEvent::OnMButtonDown (int *mX,* int *mY*) [virtual]

Definition at line 220 of file CEvent.cpp.

4.7.3.16 void CEvent::OnMButtonUp (int *mX,* int *mY*) [virtual]

Definition at line 224 of file CEvent.cpp.

4.7.3.17 void CEvent::OnMinimize ( ) `[virtual]`

Definition at line 248 of file CEvent.cpp.

4.7.3.18 void CEvent::OnMouseBlur ( ) `[virtual]`

Definition at line 192 of file CEvent.cpp.

4.7.3.19 void CEvent::OnMouseFocus ( ) `[virtual]`

Definition at line 188 of file CEvent.cpp.

4.7.3.20 void CEvent::OnMouseMove (int *mX*, int *mY*, int *relX*, int *relY*, bool *Left*, bool *Right*, bool *Middle*) `[virtual]`

Definition at line 196 of file CEvent.cpp.

4.7.3.21 void CEvent::OnMouseWheel (bool *Up*, bool *Down*) `[virtual]`

Definition at line 200 of file CEvent.cpp.

4.7.3.22 void CEvent::OnRButtonDown (int *mX*, int *mY*) `[virtual]`

Definition at line 212 of file CEvent.cpp.

4.7.3.23 void CEvent::OnRButtonUp (int *mX*, int *mY*) `[virtual]`

Definition at line 216 of file CEvent.cpp.

4.7.3.24 void CEvent::OnResize (int *w*, int *h*) `[virtual]`

Definition at line 256 of file CEvent.cpp.

4.7.3.25 void CEvent::OnRestore ( ) `[virtual]`

Definition at line 252 of file CEvent.cpp.

4.7.3.26 void CEvent::OnUser (Uint8 *type*, int *code*, void ∗ *data1*, void ∗ *data2*) `[virtual]`

Definition at line 268 of file CEvent.cpp.

The documentation for this class was generated from the following files:

- Sokoban/CEvent.h
- Sokoban/CEvent.cpp

4.8 CFPS Class Reference

`#include <CFPS.h>`

Collaboration diagram for CFPS:

Public Member Functions

- CFPS ()

  A constructor.
- void OnLoop ()

  we have our OnLoop function where all the magic happens.
- int GetFPS ()

  Get FPS.
- float GetSpeedFactor ()

  Get Speed Factor.

Static Public Attributes

- static CFPS FPSControl

4.8.1 Detailed Description

Definition at line 74 of file CFPS.h.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 CFPS::CFPS ( )

A constructor.

Set everything to 0.

Definition at line 34 of file CFPS.cpp.

4.8.3 Member Function Documentation

4.8.3.1 int CFPS::GetFPS ( )

Get FPS.

we have our two get functions that return NumFrames (FPS), and the SpeedFactor.

Definition at line 74 of file CFPS.cpp.

4.8.3.2 float CFPS::GetSpeedFactor ( )

Get Speed Factor.

we have our two get functions that return NumFrames (FPS), and the SpeedFactor.

Definition at line 82 of file CFPS.cpp.

4.8.3.3 void CFPS::OnLoop ( )

we have our OnLoop function where all the magic happens.

The first calculation we do, is finding the Frames per Second of the game. This is where SDL_GetTicks comes in. SDL_GetTicks returns the number of milliseconds that have passed since we have called SDL_Init. So, effectively, 1000 milliseconds equals 1 second. Now, notice the condition I am checking. I am checking if the OldTime (which is currently set to 0) + 1000, is less than the current tick count. Think about it, if OldTime is 0 + 1000, and 1001 milliseconds have passed, that means at least 1 second has passed. Why is that important? That tells use to restart the FPS counter.

Notice inside the if statement we then set OldTime to SDL_GetTicks, set NumFrames to the Frames count, and reset our Frames counter. So, going with the example above, OldTime now becomes 1001. Now, 1001 + 1000, is the next calculation, so another second must pass. Concurrent with this, is the Frames++ at the bottom of that function. This continuelly counts up, until the OldTime resets it. I hope that is easy to understand.

Next, we have the actual calculation for the SpeedFactor. Like I mentioned earlier, the SpeedFactor determines how fast to move objects. Now, for a better explanation. Say we are on a really slow computer that gets 1 FPS per second. That's a really bad computer by the way. We have the base movement rate set at 32 (our made up number). So, ideally, any computer will move objects 32 pixels per second. So this 1 FPS computer will move the object 32 pixels every second. Now, jump to a 10 FPS computer. We still want to move 32 pixels per second. So, that's 1/10 of 32 right? So, for each calculation we move 3.2 pixels, and after 1 second, we'll have moved 32 pixels.

To calculate this, we base it on how fast the computer is currently going. This is what LastTime is for. LastTime holds the time it took for the last loop in the game. So, in the constructor it's currently 0. Now, say we have an FPS of 10. That means we have 10 loops in the game every second. How many milliseconds is that? That's 100 milliseconds. So, 100 / 1000 (for 1 second), is 1/10. The same number we got above. Now multiply this by our desired movement rate, and we have our Speed Factor! I'll be showing you how to use this later on, just try to understand how we calculated it.

Definition at line 54 of file CFPS.cpp.

4.8.4 Member Data Documentation

4.8.4.1 CFPS CFPS::FPSControl `[static]`

whenever you make a variable of a class static, you also need to declare it elsewhere. Notice I have CFPS::FPSControl at the top. If you fail to do this, you'll get an undeclared error.

Definition at line 76 of file CFPS.h.

The documentation for this class was generated from the following files:

- Sokoban/CFPS.h
- Sokoban/CFPS.cpp

4.9 CMap Class Reference

`#include <CMap.h>`

Public Member Functions

- CMap ()

  A constructor.
- bool OnLoad (char ∗File)

  Next we have the OnLoad function.
- void OnRender (SDL_Surface ∗Surf_Display, int MapX, int MapY)

  Next, we have the function to render a map.
- CTile ∗ GetTile (int X, int Y)

Public Attributes

- SDL_Surface ∗ Surf_Tileset

4.9.1 Detailed Description

Definition at line 40 of file CMap.h.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 CMap::CMap ( )

A constructor.

The constructor sets the Tileset to NULL, obviously.

Definition at line 7 of file CMap.cpp.

4.9.3 Member Function Documentation

4.9.3.1 CTile ∗ CMap::GetTile (int *X,* int *Y*)

Definition at line 77 of file CMap.cpp.

4.9.3.2 bool CMap::OnLoad (char ∗ *File*)

Next we have the OnLoad function.

First, we are clearing out any old tiles, that way if we load twice, it won't have double the amount of tiles, effectively loading a new map. We then open up a FileHandle, and try to open the requested map file. Now, we go through the map file and grab each tile. This is accomplished by using the two loops. The outer loop is the Y axis loop, going from the top row to the bottom row of the map file. The inner loop is the X axis loop, going from the left most tile, to the right most tile. Pay attention to how this loop works, because it is also used below in the OnRender function. It basically goes from the top left tile, to the bottom right tile, 1 tile at a time. Inside of those loops we create a temporary tile, and load the file information into it. We then push that into our TileList, effectively saving the tile. We close the filehandle, and we are done.

Definition at line 15 of file CMap.cpp.

4.9.3.3 void CMap::OnRender (SDL_Surface ∗ *Surf_Display,* int *MapX,* int *MapY*)

Next, we have the function to render a map.

Notice the MapX and MapY arguments. These tell use where to render this map on the screen. This is useful later on for moving maps around. First in the function we check for a valid tileset, because we are going to access this tileset directly and don't want to cause a crash. We then grab the TilesetWidth and TilesetHeight in Tiles. This is important, because we need to know how many tiles a tileset contains, not its actual width and height. That way we can match up a TileID to a tileset. So, a Tileset with a Width and Height, in tiles, of 2x2, would have 4 tiles on it, but in reality it would be 32x32 pixels. So, a TileID of 0 would match the first tile, a TileID of 1 would be the next, and so on. TileIDs repeat on the next row, from left to right.

After that, we do something with the Tileset. What we are doing is calculating where on the Tileset to grab the appropriate tile. This is by grabing the TileID of the tile first, and then converting that to a Tile coordinate. A little bit of explanation here. We have our 2x2 tileset, and a TileID of 1. Figuring out X, we would get 1 % 2, which would be 1 still. We when multiply that by the TILE_SIZE, and get 16. That is the correct X coordinate for the Tile. Same with the Y, we put 1 / 2, which is 0.5. Since this an integer operation, the .5 is automatically dropped. Thus we are left with 0. Which is also the correct row. Now, say we had a TileID of 2. 2 % 2 = 0, and 2 / 2 = 1. See how the X repeats in a pattern? 0, 1, 0, 1... And notice how Y increases every time it goes past the Tileset Width? 0, 0, 1, 1. I hope this is clear, as it is somewhat hard to explain.

Next, we actually draw the tile to the screen using the coordinates we just calculated, and then increase the ID to go to the next tile. A little side note here, we could, for the sake of speed, create an OnRender_Cache function that would perform this same operation, but would draw to a Surface defined in the CMap class. Something like SDL_-Surface∗ Surf_Map. Then, the OnRender function would render the Surf_Map only, and not perform any operations. But also take note that that method does not necessarely work later on when we want to animate tiles.

Definition at line 49 of file CMap.cpp.

4.9.4 Member Data Documentation

4.9.4.1 SDL_Surface∗ CMap::Surf_Tileset

Definition at line 42 of file CMap.h.

The documentation for this class was generated from the following files:

- Sokoban/CMap.h
- Sokoban/CMap.cpp

4.10 CPlayer Class Reference

`#include <CPlayer.h>`

Inheritance diagram for CPlayer:

Collaboration diagram for CPlayer:

Public Member Functions

- CPlayer ()
- bool OnLoad (char ∗File, int Width, int Height, int MaxFrames, SDL_Surface ∗Screen_Display)
- void OnLoop ()

  On Loop.
- void OnRender (SDL_Surface ∗Surf_Display)

  On Render.
- void OnCleanup ()
- void OnAnimate ()

  On Animate.
- bool OnCollision (CEntity ∗Entity)

  On Collision.

Additional Inherited Members

4.10.1 Detailed Description

Definition at line 6 of file CPlayer.h.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 CPlayer::CPlayer ( )

Definition at line 3 of file CPlayer.cpp.

4.10.3 Member Function Documentation

4.10.3.1 void CPlayer::OnAnimate ( ) [virtual]

On Animate.

This is fairly straight forward. Like I said above, depending on which direction we are moving, we set the animation.

Reimplemented from CEntity.

Definition at line 26 of file CPlayer.cpp.

4.10.3.2 void CPlayer::OnCleanup ( ) [virtual]

Reimplemented from CEntity.

Definition at line 22 of file CPlayer.cpp.

4.10.3.3 bool CPlayer::OnCollision (CEntity ∗ *Entity*) [virtual]

On Collision.

Yay, an empty function. This function will be called whenever two entities hit each other. This entire time we've been creating an class for basic entities. In order to take advantage of this function, you will need to create our own class (like CPlayer, which we'll do later on in this tutorial) and override this function. This is because the action taken by an entity depends upon it's type. Like my earlier examples, a Goomba will not react the same way to a collision with a spike as Mario would.

Reimplemented from CEntity.

Definition at line 36 of file CPlayer.cpp.

4.10.3.4 bool CPlayer::OnLoad (char ∗ *File,* int *Width,* int *Height,* int *MaxFrames,* SDL_Surface ∗ *Screen_Display*) [virtual]

Reimplemented from CEntity.

Definition at line 6 of file CPlayer.cpp.

4.10.3.5 void CPlayer::OnLoop ( ) [virtual]

On Loop.

Lots of new things here. First, remember MoveLeft and MoveRight are used to indicate if the entity is moving left or right (duh). If both are these are false, we call StopMove, which will handle settings Acceleration in the reverse direction so we come to a gradual stop. Below that, if MoveLeft or MoveRight are true, we give AccelX a value. 0.5 is an arbitrary number, you probably want to mess with this until you find a value that you like. Just below that, we have a condition checking if gravity is turned on for that particular entity. If it is, we set AccelY. This will cause our entity to fall down. Again, this is whatever value you want. The larger the number, the faster you will fall. Next, below that we have our SpeedX and SpeedY. Again, we increase the Speed of the entity over time in correlation to our Speed Factor. I want my player to increase in speed 1 pixel per second. This would be:

Reimplemented from CEntity.

Definition at line 14 of file CPlayer.cpp.

4.10.3.6 void CPlayer::OnRender (SDL_Surface ∗ *Surf_Display*) [virtual]

On Render.

First, we have added the Camera X and Y to the rendering coordinates for the entity. This makes the entities render in connection to where the camera is. This way, if the camera moves, all the entities will in respect to the camera position. Next, we have added CurrentFrameCol and CurrentFrameRow to the X2 and Y2 of OnDraw. This makes it so we can set different animations.

The first column has Yoshi facing the left, while the second column has Yoshi facing to the right. When we move to the left, we will want to set CurrentFrameCol = 0, so that Yoshi appears to be walking left. And when we move to the right, we will set CurrentFrameCol = 1, so that Yoshi appears to be walking to the right.

Reimplemented from CEntity.

Definition at line 18 of file CPlayer.cpp.

The documentation for this class was generated from the following files:

- Sokoban/CPlayer.h
- Sokoban/CPlayer.cpp

4.11 CSoundBank Class Reference

```
#include <CSoundBank.h>
```

Collaboration diagram for CSoundBank:

Public Member Functions

- CSoundBank ()
- int OnLoad (char ∗File)
- void OnCleanup ()
- void Play (int ID)

Public Attributes

- std::vector< Mix_Chunk ∗ > SoundList

Static Public Attributes

- static CSoundBank SoundControl

4.11.1 Detailed Description

Definition at line 8 of file CSoundBank.h.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 CSoundBank::CSoundBank ( )

Definition at line 5 of file CSoundBank.cpp.

4.11.3 Member Function Documentation

4.11.3.1 void CSoundBank::OnCleanup ( )

Definition at line 20 of file CSoundBank.cpp.

4.11.3.2 int CSoundBank::OnLoad (char ∗ *File*)

Definition at line 8 of file CSoundBank.cpp.

4.11.3.3 void CSoundBank::Play (int *ID*)

Definition at line 28 of file CSoundBank.cpp.

4.11.4 Member Data Documentation

4.11.4.1 CSoundBank CSoundBank::SoundControl [static]

Definition at line 10 of file CSoundBank.h.

4.11.4.2 std::vector<Mix_Chunk∗> CSoundBank::SoundList

Definition at line 12 of file CSoundBank.h.

The documentation for this class was generated from the following files:

- Sokoban/CSoundBank.h
- Sokoban/CSoundBank.cpp

4.12 CSurface Class Reference

A surface blitting class.

`#include <CSurface.h>`

Public Member Functions

- CSurface ()

  A constructor.

Static Public Member Functions

- static SDL_Surface ∗ OnLoad (char ∗File, SDL_Surface ∗Screen_Display)

  Load image in memory.
- static bool OnDraw (SDL_Surface ∗Surf_Dest, SDL_Surface ∗Surf_Src, int X, int Y)

  Draw surfaces onto other surfaces.
- static bool OnDraw (SDL_Surface ∗Surf_Dest, SDL_Surface ∗Surf_Src, int X, int Y, int X2, int Y2, int W, int H)

  Draw a part of surfaces onto other surfaces.
- static bool Transparent (SDL_Surface ∗Surf_Dest, int R, int G, int B)

  Set transparent key color on BMP images.

4.12.1 Detailed Description

A surface blitting class.

A more elaborate class description.

Definition at line 50 of file CSurface.h.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 CSurface::CSurface ( )

A constructor.

A more elaborate description of the constructor in CPP.

Definition at line 15 of file CSurface.cpp.

4.12.3 Member Function Documentation

4.12.3.1 bool CSurface::OnDraw (SDL_Surface ∗ *Surf_Dest,* SDL_Surface ∗ *Surf_Src,* int *X,* int *Y*) `[static]`

Draw surfaces onto other surfaces.

The start of the function makes sure we have valid surfaces, if we don't, return false. Next, we find SDL_Rect. This is a SDL structure that basically has four members: x, y, w, h. This, of course, creates the dimensions for a rectangle. We are only worried about where we are drawing to, not the size. So we assign X, Y coordinates to the destination surface. If you are wondering what NULL is within SDL_BlitSurface, it's another parameter for a SDL_Rect. We'll get to this later on in this lesson. < Test valid surfaces.

< creates the dimensions for a rectangle.

< blit surface

Definition at line 41 of file CSurface.cpp.

4.12.3.2 bool CSurface::OnDraw (SDL_Surface ∗ *Surf_Dest,* SDL_Surface ∗ *Surf_Src,* int *X,* int *Y,* int *X2,* int *Y2,* int *W,* int *H*) `[static]`

Draw a part of surfaces onto other surfaces.

Notice that it's basically the same function as our first one, except we've added another SDL_Rect. This source rect allows use to specify what pixels from the source to copy over to the destination. If we specified 0, 0, 50, 50 as parameters for X2...H, it would only draw upper left part of the surface (a 50x50 square).

Definition at line 61 of file CSurface.cpp.

4.12.3.3 SDL_Surface ∗ CSurface::OnLoad (char ∗ *File,* SDL_Surface ∗ *Screen_Display*) `[static]`

Load image in memory.

There are a couple of important things to note here. Firstly, always remember that when you make a pointer to set it to NULL, or 0. Many problems can come along later if you fail to do this. Secondly, notice how SDL_DisplayFormat returns a new Surface, and doesn't overwrite the original. This important to remember because since it creates a new surface, we have to free the old one. Otherwise, we have a surface floating around in memory.

Remember I said BMPs don't support alpha layers? Well, PNGs do! < pointer to set it to NULL, or 0.

< Call SDL method to load images.

< returns a new Surface.

Definition at line 24 of file CSurface.cpp.

4.12.3.4 bool CSurface::Transparent (SDL_Surface ∗ *Surf_Dest,* int *R,* int *G,* int *B*) `[static]`

Set transparent key color on BMP images.

This function first checks to see if we have a valid surface. If so, we set a color key (transparency) for a color. The first argument is the surface to apply the color key to, the second is some flags telling SDL how to perform the operation, and the third is the color to make transparent. The flags being applied are basic, the first tells SDL to apply the color key to the source (the surface being passed) and the second tells SDL to try to use RLE acceleration (basically, try to make drawing later on faster). The third argument is a little bit more complex; we are using SDL_-MapRGB in order to create a color. SDL_MapRGB takes a surface, and your requested color (R, G, B), and tries to match it as close as it can to that surface. You might be thinking why this is useful. Not all surfaces have the same color palette. Remember the old NES days where there was only a few colors that could be used? Same idea here, SDL_MapRGB takes a color and matches it with the closest color on that surface palette.

Definition at line 87 of file CSurface.cpp.

The documentation for this class was generated from the following files:

- Sokoban/CSurface.h
- Sokoban/CSurface.cpp

4.13 CTile Class Reference

This class is going to define a single tile on a map.

```
#include <CTile.h>
```

Public Member Functions

- CTile ()

  A tile constructor.

Public Attributes

- int TileID
- int TypeID

4.13.1 Detailed Description

This class is going to define a single tile on a map.

In gaming terms, a tile is basically a single square graphic that we draw to the screen. Remember the animation tutorial, each frame of Yoshi could be thought of as a tile. So when I say we have a map made up of many tiles, we are taking these tiles and repeating them in a grid sequence.

And since we have a single graphic containg all of our tiles, we only have to load one graphic, and not several for each tile. Each tile then needs some properties. The most obvious is which graphical tile to use.

Definition at line 41 of file CTile.h.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 CTile::CTile ( )

A tile constructor.

Sets IDs to zero.

Definition at line 13 of file CTile.cpp.

4.13.3 Member Data Documentation

4.13.3.1 int CTile::TileID

Definition at line 43 of file CTile.h.

4.13.3.2 int CTile::TypeID

Definition at line 44 of file CTile.h.

The documentation for this class was generated from the following files:

- Sokoban/CTile.h
- Sokoban/CTile.cpp

Chapter 5

File Documentation

5.1 Sokoban/CAnimation.cpp File Reference

```
#include "CAnimation.h"
```

Include dependency graph for CAnimation.cpp:

5.2 Sokoban/CAnimation.h File Reference

Header definitions for the animation part.

```
#include <SDL.h>
```

Include dependency graph for CAnimation.h: This graph shows which files directly or indirectly include this file:

Classes

- class CAnimation

  A SDL Animation class.

5.2.1 Detailed Description

Header definitions for the animation part. Details.

Author

Petar Jerčić

Definition in file CAnimation.h.

5.3 Sokoban/CApp.cpp File Reference

The core of our program.

```
#include "CApp.h"
```

Include dependency graph for CApp.cpp:

Functions

- int main (int argc, char ∗argv[])

5.3.1 Detailed Description

The core of our program. The CApp class is setting the stage for our entire program. Let me step aside to take special note of how games are typically setup. Most games consist of 5 functions that handle how the game processes. These processes are typically:

Initialize This function handles all the loading of data, whether it be textures, maps, NPCs, or whatever.

Event This function handles all input events from the mouse, keyboard, joysticks, or other devices.

Loop This function handles all the data updates, such as a NPCs moving across the screen, decreasing your health bar, or whatever

Render This function handles all the rendering of anything that shows up on the screen. It does NOT handle data manipulation, as this is what the Loop function is supposed to handle.

Cleanup This function simply cleans up any resources loaded, and insures a peaceful quitting of the game.

It's important to understand that games are one gigantic loop. Within this loop we find events, update data, and render pictures.

Definition in file CApp.cpp.

5.3.2 Function Documentation

5.3.2.1 int main (int *argc*, char ∗ *argv[]*)

Definition at line 98 of file CApp.cpp.

5.4 Sokoban/CApp.h File Reference

Header definitions for the core of our program.

```
#include <SDL.h>
#include <stdio.h>
#include "CSurface.h"
#include "CEvent.h"
#include "CAnimation.h"
#include "CEntity.h"
#include "Define.h"
#include "CArea.h"
#include "CCamera.h"
#include "CPlayer.h"
#include "CSoundBank.h"
```

Include dependency graph for CApp.h: This graph shows which files directly or indirectly include this file:

Classes

- class CApp

  A core program class.

5.4.1 Detailed Description

Header definitions for the core of our program. Details.

Details.

Author

Petar Jerčić

Definition in file CApp.h.

5.5 Sokoban/CApp_OnCleanup.cpp File Reference

```
#include "CApp.h"
```
Include dependency graph for CApp_OnCleanup.cpp:

5.6 Sokoban/CApp_OnEvent.cpp File Reference

Header definitions for the core of our program.

```
#include "CApp.h"
```
Include dependency graph for CApp_OnEvent.cpp:

5.6.1 Detailed Description

Header definitions for the core of our program. Details.

Author

Petar Jerčić

Definition in file CApp_OnEvent.cpp.

5.7 Sokoban/CApp_OnInit.cpp File Reference

```
#include "CApp.h"
```
Include dependency graph for CApp_OnInit.cpp:

5.8 Sokoban/CApp_OnLoop.cpp File Reference

Code for our big loop.

```
#include "CApp.h"
```
Include dependency graph for CApp_OnLoop.cpp:

5.8.1 Detailed Description

Code for our big loop. Details.

Author

Petar Jerčić

Definition in file CApp_OnLoop.cpp.

5.9 Sokoban/CApp_OnRender.cpp File Reference

```
#include "CApp.h"
```
Include dependency graph for CApp_OnRender.cpp:

5.10 Sokoban/CArea.cpp File Reference

```
#include "CArea.h"
```
Include dependency graph for CArea.cpp:

5.11 Sokoban/CArea.h File Reference

```
#include "CMap.h"
```
Include dependency graph for CArea.h: This graph shows which files directly or indirectly include this file:

Classes

- class CArea

5.12 Sokoban/CCamera.cpp File Reference

code for the camera.

```
#include "CCamera.h"
```
Include dependency graph for CCamera.cpp:

5.12.1 Detailed Description

code for the camera. From top to bottom, like usual, we have are static member first. We then have the constructor defaulting some variables. The OnMove function will increase the X and Y, like I said before. We then have GetX. It will check to see if we have a valid target, and if so return the targets coordinates as the camera coordinates, otherwise return the camera x. For centering mode, we take the screen width and divide by two to find the center of the screen, and then subtract it from the target coordinates. This will center the camera to the target. We then have the SetPos and SetTarget functions, which are self-explanatory.

Author

Petar Jerčić

Definition in file CCamera.cpp.

5.13 Sokoban/CCamera.h File Reference

Header definitions for the camera.

```
#include <SDL.h>
#include "Define.h"
```
Include dependency graph for CCamera.h: This graph shows which files directly or indirectly include this file:

Classes

- class CCamera

Enumerations

- enum { TARGET_MODE_NORMAL = 0, TARGET_MODE_CENTER }

5.13.1 Detailed Description

Header definitions for the camera. First, we have a control member object, just like area. Then we have the coordinates of where the camera is. An extra thing I threw in, was the ability to target something. For example, in a Megaman game the camera would target Megaman himself. That way, when Megaman moves the camera would automatically update. So we have two pointers for X and Y coordinates. If these are null, the camera will automatically revert to the cameras position. Next, we have a target mode, which, for now, is either normal (the camera will position to the top left of the target) or center (will center the camera to the target). Pretty simply I think.

We then have a few functions, the first is the OnMove, which will increase the X and Y of the camera by MoveX and MoveY. So OnMove(-1, 0) would move the camera to the left one pixel. We then have Get functions for the coordinates, and the ability to set the coordinates and set a target.

Author

Petar Jerčić

Definition in file CCamera.h.

5.13.2 Enumeration Type Documentation

5.13.2.1 anonymous enum

Enumerator:

TARGET_MODE_NORMAL

TARGET_MODE_CENTER

Definition at line 18 of file CCamera.h.

5.14 Sokoban/CEntity.cpp File Reference

Header definitions for the entities of our program.

```
#include "CEntity.h"
```

Include dependency graph for CEntity.cpp:

5.14.1 Detailed Description

Header definitions for the entities of our program. Details.

Author

Petar Jerčić

Definition in file CEntity.cpp.

5.15 Sokoban/CEntity.h File Reference

Header definitions for the entities of our program.

```
#include <vector>
#include "CAnimation.h"
#include "CSurface.h"
#include "CArea.h"
#include "CCamera.h"
#include "CFPS.h"
```

Include dependency graph for CEntity.h: This graph shows which files directly or indirectly include this file:

Classes

- class CEntity

  A entity class.
- class CEntityCol

Enumerations

- enum { ENTITY_TYPE_GENERIC = 0, ENTITY_TYPE_PLAYER }

  An enum.
- enum { ENTITY_FLAG_NONE = 0, ENTITY_FLAG_GRAVITY = 0x00000001, ENTITY_FLAG_GHOST = 0x00000002, ENTITY_FLAG_MAPONLY = 0x00000004 }

  An enum.

5.15.1 Detailed Description

Header definitions for the entities of our program. Okay, now for some basic explanation. What we are doing here is encapsulating the basic 5 components I mentioned within the first lesson (excluding Events, which will be handled in a later lesson). This allows us to handle Entities within the game much more easily, rather than clumping them together with everything else in the game within the main CApp class. This will also be the way we handle other things as well.

Now, lets go ahead and add this new Player to the game. But first, some explanation. Remember, anything that moves in the game or that can be interacted with (besides the Map), is an Entity. So, our main Player will be an entity. But, Player is distinct, he likes certain things other Entities do not. So, instead of simply doing CEntity Player; we need to create a class just for Player. But! Before we can even to get to making the Player class, we're going to need to extend the Entity class.

Author

Petar Jerčić

Definition in file CEntity.h.

5.15.2 Enumeration Type Documentation

5.15.2.1 anonymous enum

An enum.

You'll notice we have another enum, like the Flags, to specify the type of entity. This is important when dealing with collision events. So, back to the Mario and Spike example; lets say a Goomba happens to step in a spike. Now, in Mario world, the Goomba would not die. So, the Spike needs to know that entities type. Simple enough I think.

Enumerator:

ENTITY_TYPE_GENERIC

ENTITY_TYPE_PLAYER

Definition at line 136 of file CEntity.h.

5.15.2.2 anonymous enum

An enum.

Now, last for movement, we have MoveLeft, MoveRight, Flags, and StopMove. Now, MoveLeft and MoveRight are simply flags, true or false, that determine if the entity is moving left or right. If true, the AccelX will be set a value, causing the entity to move. Now, StopMove simply sets the AccelX in the opposite direction until we reach 0. Again, we don't stop suddenly in real life, we gradually stop. The last item, Flags, is used to determine a few things.

Enumerator:

ENTITY_FLAG_NONE

ENTITY_FLAG_GRAVITY

ENTITY_FLAG_GHOST

ENTITY_FLAG_MAPONLY

Definition at line 144 of file CEntity.h.

5.16 Sokoban/CEntityCol.cpp File Reference

```
#include "CEntity.h"
```
Include dependency graph for CEntityCol.cpp:

5.17 Sokoban/CEvent.cpp File Reference

The core of our program.

```
#include "CEvent.h"
```
Include dependency graph for CEvent.cpp:

5.17.1 Detailed Description

The core of our program. The CApp class is setting the stage for our entire program. Let me step aside to take special note of how games are typically setup. Most games consist of 5 functions that handle how the game processes. These processes are typically:

Initialize This function handles all the loading of data, whether it be textures, maps, NPCs, or whatever.

Event This function handles all input events from the mouse, keyboard, joysticks, or other devices.

Loop This function handles all the data updates, such as a NPCs moving across the screen, decreasing your health bar, or whatever

Render This function handles all the rendering of anything that shows up on the screen. It does NOT handle data manipulation, as this is what the Loop function is supposed to handle.

Cleanup This function simply cleans up any resources loaded, and insures a peaceful quitting of the game.

It's important to understand that games are one gigantic loop. Within this loop we find events, update data, and render pictures.

Definition in file CEvent.cpp.

5.18 Sokoban/CEvent.h File Reference

Header definitions for the core of our program.

```
#include <SDL.h>
```

Include dependency graph for CEvent.h: This graph shows which files directly or indirectly include this file:

Classes

- class CEvent

  An event class.

5.18.1 Detailed Description

Header definitions for the core of our program. Details.

Author

Petar Jerčić

Definition in file CEvent.h.

5.19 Sokoban/CFPS.cpp File Reference

The core of our program.

```
#include "CFPS.h"
```

Include dependency graph for CFPS.cpp:

5.19.1 Detailed Description

The core of our program. The CApp class is setting the stage for our entire program. Let me step aside to take special note of how games are typically setup. Most games consist of 5 functions that handle how the game processes. These processes are typically:

Initialize This function handles all the loading of data, whether it be textures, maps, NPCs, or whatever.

Event This function handles all input events from the mouse, keyboard, joysticks, or other devices.

Loop This function handles all the data updates, such as a NPCs moving across the screen, decreasing your health bar, or whatever

Render This function handles all the rendering of anything that shows up on the screen. It does NOT handle data manipulation, as this is what the Loop function is supposed to handle.

Cleanup This function simply cleans up any resources loaded, and insures a peaceful quitting of the game.

It's important to understand that games are one gigantic loop. Within this loop we find events, update data, and render pictures.

Author

Petar Jerčić

Definition in file CFPS.cpp.

5.20 Sokoban/CFPS.h File Reference

Header definitions for the frame rate of our program.

```
#include <SDL.h>
```

Include dependency graph for CFPS.h: This graph shows which files directly or indirectly include this file:

Classes

- class CFPS

5.20.1 Detailed Description

Header definitions for the frame rate of our program. But before we get into that, we're going to need a standard for movement. You see, when we play our games on old Pentium 2 as opposed to a quad core, there are apparent differences in speed. So, if we did a simple X++; on the older machine, the entity would move slower than the newer machine. So the idea is simple, figure out the frame rate of the machine, and base our movements off of the frame rate. I call that the speed factor. Now, in order for this to work, we have to have a base frame rate. I usually pick 32, as that is a good number. So if our machine had a frame rate of 32, our speed factor would be 1. If our frame rate was 64, our speed factor would be 0.5. We move twice as slower on the faster machine, that way speeds are consistent across different machines.

Author

Petar Jerčić

Definition in file CFPS.h.

5.21 Sokoban/CMap.cpp File Reference

```
#include "CMap.h"
```

Include dependency graph for CMap.cpp:

5.22 Sokoban/CMap.h File Reference

```
#include <SDL.h>
#include <vector>
#include "CTile.h"
#include "CSurface.h"
```

Include dependency graph for CMap.h: This graph shows which files directly or indirectly include this file:

Classes

- class CMap

5.23 Sokoban/CPlayer.cpp File Reference

```
#include "CPlayer.h"
```

Include dependency graph for CPlayer.cpp:

5.24 Sokoban/CPlayer.h File Reference

```
#include "CEntity.h"
```
Include dependency graph for CPlayer.h: This graph shows which files directly or indirectly include this file:

Classes

- class CPlayer

5.25 Sokoban/CSoundBank.cpp File Reference

```
#include "CSoundBank.h"
```
Include dependency graph for CSoundBank.cpp:

5.26 Sokoban/CSoundBank.h File Reference

```
#include <SDL.h>
#include <SDL_mixer.h>
#include <vector>
```
Include dependency graph for CSoundBank.h: This graph shows which files directly or indirectly include this file:

Classes

- class CSoundBank

5.27 Sokoban/CSurface.cpp File Reference

```
#include "CSurface.h"
```
Include dependency graph for CSurface.cpp:

5.28 Sokoban/CSurface.h File Reference

```
#include <SDL.h>
#include <SDL_image.h>
#include <stdio.h>
```
Include dependency graph for CSurface.h: This graph shows which files directly or indirectly include this file:

Classes

- class CSurface

  A surface blitting class.

5.29 Sokoban/CTile.cpp File Reference

Code for the tiles.

```
#include "CTile.h"
```
Include dependency graph for CTile.cpp:

5.29.1 Detailed Description

Code for the tiles.

Author

Petar Jerčić

Definition in file CTile.cpp.

5.30 Sokoban/CTile.h File Reference

Code for the area.

`#include "Define.h"`

Include dependency graph for CTile.h: This graph shows which files directly or indirectly include this file:

Classes

- class CTile
  This class is going to define a single tile on a map.

Enumerations

- enum { TILE_TYPE_NONE = 0, TILE_TYPE_NORMAL, TILE_TYPE_BLOCK }
  This lets us assign or check the TypeID, and know what it is at the same time.

5.30.1 Detailed Description

Code for the area. This file is going to hold some constant values, as defines, that will define things like map width and height, screen width and height, and tile size.

Header definitions for the tiles.

Header definitions for the maps.

Header definitions for the area.

Details.

Author

Petar Jerčić

What we are going to do is have one Area object, this Area object will have many children Map objects in a vector (like the Entities in a vector). Within each of these Map objects, they will have a vector of each individual tile. So basically, we have a bunch of tiles that make up a map, and a bunch of maps make up an area.

Okay, cool! I know that is quite a bit to take in at once, but don't fret! It's really not too bad once you get the hang of it. I want to say that right now we could create Map objects and render maps just fine. This would be ideal for games like Mario and Megaman, as they are just one big map. But for games like Zelda and Metroid, it won't work too well. That is where Areas come in. Just like maps having file formats, each area will have its own file and file format. Each area is going to have a tileset to load, the size of the area, and the maps that are to be loaded in each area. Here is the area we are going to be using below:

./tilesets/1.png 3 ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map ./maps/1.map

Save this into a file called 1.area, and save it in your maps directory. Just like maps, this area will tile maps.

Author

Petar Jerčić

What we are going to do is have one Area object, this Area object will have many children Map objects in a vector (like the Entities in a vector). Within each of these Map objects, they will have a vector of each individual tile. So basically, we have a bunch of tiles that make up a map, and a bunch of maps make up an area.

The first thing we need to consider is that our maps will be text files. So we need to come up with a file format, that way we can easily edit our maps outside of our code. And possibly later you could create a map editor.

Each map is going to have a width and height, defining the number of tiles. So a map of 10x10, would have 100 tiles in it. We are going to make all of our maps the same width and height, so we don't have to define that within the map files (MAP_WIDTH and MAP_HEIGHT). The part that we do need to add though is the TileID and TypeID for each individual tile. We are going to use the file format that I came up with, as it is relatively simple. Take a look at the 5x5 map example below:

0:0 0:0 0:0 0:0 0:0 1:0 1:0 1:0 0:0 0:0 1:0 1:0 1:0 0:0 0:0 1:0 1:0 1:0 0:0 0:0 1:0 1:0 1:0 0:0 0:0

Each tile within the file would consist of 0:0, effectively being the TileID:TypeID. A space would be the deliminator between the tiles.

First create a folder in the same directory as your exe called maps. This folder is where all your maps will be. Save that map as 1.map. Also, make another folder called tilesets in the same directory as your exe, and save the tileset above as 1.png.

Author

Petar Jerčić

What we are going to do is have one Area object, this Area object will have many children Map objects in a vector (like the Entities in a vector). Within each of these Map objects, they will have a vector of each individual tile. So basically, we have a bunch of tiles that make up a map, and a bunch of maps make up an area.

Author

Petar Jerčić

These are going to be some values that we will be using throughout making our Maps and such.

Author

Petar Jerčić

Definition in file CTile.h.

5.30.2 Enumeration Type Documentation

5.30.2.1 anonymous enum

This lets us assign or check the TypeID, and know what it is at the same time.

So if I did TypeID == TILE_TYPE_BLOCK, I would be able to notice in my code that I am checking for a block type of tile. TypeID == 2 isn't as easy to interpret. If you never used enums, think of them like const variables. The value never changes. Also, the starting value defines the rest, notice I have = 0 on TILE_TYPE_NONE. From there the rest of the variables would automatically be assigned values in increments of 1.

Enumerator:

TILE_TYPE_NONE
TILE_TYPE_NORMAL
TILE_TYPE_BLOCK

Definition at line 28 of file CTile.h.

5.31 Sokoban/Define.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define MAP_WIDTH 40
- #define MAP_HEIGHT 40
- #define TILE_SIZE 16
- #define SCREEN_WIDTH 640
- #define SCREEN_HEIGHT 480

5.31.1 Macro Definition Documentation

5.31.1.1 #define MAP_HEIGHT 40

Definition at line 13 of file Define.h.

5.31.1.2 #define MAP_WIDTH 40

Definition at line 12 of file Define.h.

5.31.1.3 #define SCREEN_HEIGHT 480

Definition at line 19 of file Define.h.

5.31.1.4 #define SCREEN_WIDTH 640

Definition at line 18 of file Define.h.

5.31.1.5 #define TILE_SIZE 16

Definition at line 15 of file Define.h.

Index