

The MiniGame Project

Contents

The MiniGame Project	1
1. Objective	2
2. The project.....	2
.....MiniGame	
.....	2
3.	2
3.1. PowerUps	2
3.1.1. Create the project.....	3
3.1.2. General.....	3
3.1.3. PowerUp class.....	3
3.1.4. Test the class.....	5
3.2. PowerUpManager	6
3.2.1. Project.....	6
3.2.2. General.....	6
3.2.3. PowerUpManager class	6
3.2.4. Test the PowerUpManager class	7
3.3. MiniGame Part 1	8
3.3.1. Create the project.....	8
3.3.2. Level class.....	8
3.3.3. Test the Level class.....	9
3.3.4. Avatar class.....	10
3.4. MiniGame Part 2	15
3.4.1. Animated Avatar class	15
3.4.2. Flip the avatar.....	17
3.4.3. Load level vertices from svg file	17
3.4.4. Camera – part1	18
3.4.5. Camera – part2	19
3.5. MiniGame Part 3	20
3.5.1. Platform.....	20
3.5.2. End of the game.....	21
3.6. MiniGame Part 4 - HUD	23
3.6.1. The HUD	23
3.7. MiniGame Part 5 - Sound.....	25

3.7.1.	Introduction exercise.....	25
3.7.2.	Power up hit sound	26
4.	Submission instructions	26

1. Objective

At the end of these exercises you should be able to implement the minigame.
Including:

- Sprite animations
- Texture transformations
- Gravity
- Collision detection
- Ray casting
- Camera transformations
- Heads up display
- Sounds

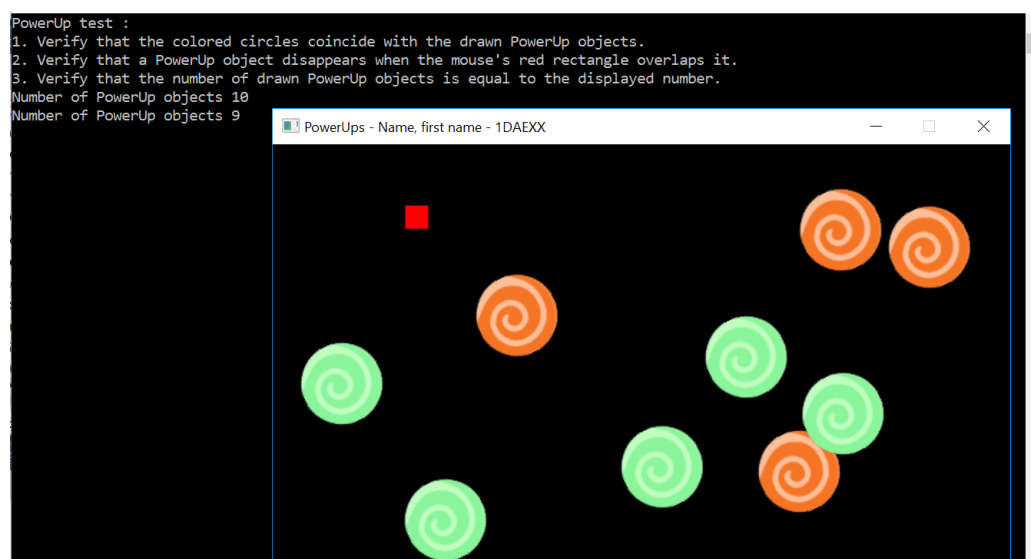
2. The solution

Create an **empty solution** with name (G|S)DxxMiniGameNameFirstname where GDxx or SDxx is your group name.

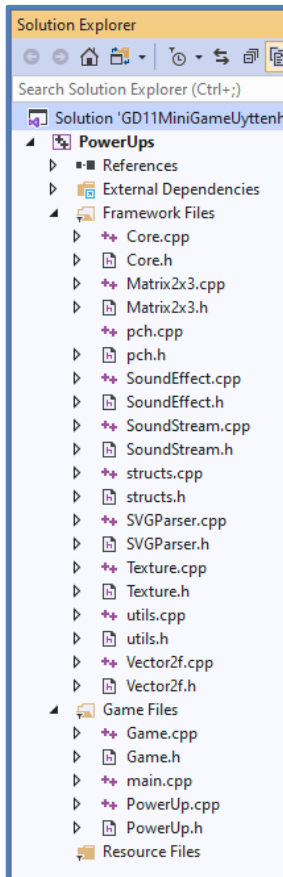
Add the libraries folder and both the props files to the **solution** folder.

3. MiniGame

3.1. PowerUps



3.1.1. Create the project



Add a new **classes framework project** with name **PowerUps** to the empty solution and set it as StartUp project.

Delete the generated file **PowerUps.cpp**.

Rename the filters to "**Framework Files**" and "**Game Files**". Move the pch files into the framework files filter.

Copy the framework files to this project folder using windows explorer, add them to the project in visual studio to the framework filter. Put Game and main in the Game files filter.

The **window title**: it should mention the name of the project, your name, first name and group.

3.1.2. General

In this project you will implement the **PowerUp** class. It shows a red or brown rotating circular image and can be hit by a rectangular shape.

This assignment contains a Game class that you can use to test the correct working of your PowerUp class once it is finished.

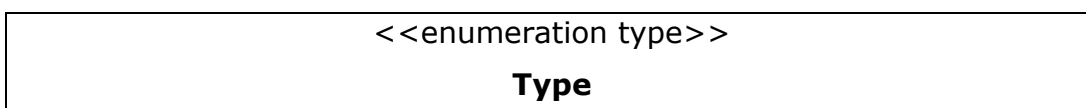
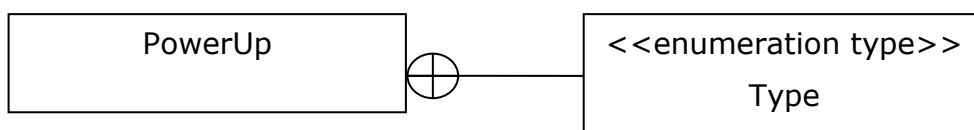
This class is the first part of a **small game** that you will further elaborate in the coming labs.

3.1.3. PowerUp class

The file PowerUp.h is given. See Codefiles / 01_PowerUpTest

The image is also given, see Resources. Copy PowerUp.png to a resources/images folder you create in the project folder.

a. The UML class diagram



```
green = 1
brown = 2
```

PowerUp

```
- m_Type: const Type
- m_Shape: Circlef
- m_pTexture: const Texture*
- m_TextClip: Rectf
- m_Angle: float
- m_RotSpeed: const float = 360

+ PowerUp(center: const Point2f&, type: PowerUp::Type )
+ Update( float elapsedSec ): void
+ Draw( ): void {query}
+ IsOverlapping(rect : const Rectf&): bool {query}
```

b. Description of some data members

m_Type

Indicates the type of the power up, green or brown.

m_Shape

Describes the shape and position of the power up. Will be used to decide whether another shape overlaps the power up and to draw the power up at the right position.

m_TextClip

The rectangular clip indicating which part of the texture is used to draw it, depends on the type. The picture below shows the clip for the brown power up.



```
left      0.0f
bottom    m_Texture.GetHeight( )
width     m_Texture.GetWidth( )
height    m_Texture.GetHeight( ) / 2
```

m_RotSpeed

The power up rotates round its center with a rotational speed as indicated by this member.

Let the power up make one complete rotation (360 degrees) per second

m_Angle

This angle is updated in the Update method using the rotational speed and the elapsed seconds.

You need it in the Draw method to set the OpenGL rotation matrix

c. Description of the methods

Constructor
Initializes the data members
Update
Updates the angle using the rotation speed and elapsed seconds
Draw
Indicates which model matrix OpenGL has to be used and draws the clip of the sprite sheet.
IsOverlapping
Returns true when the rectangular shape overlaps the circular shape of the power up. Tip: use <code>utils::IsOverlapping</code> function

3.1.4. Test the class

Replace the files **Game.h and cpp** by the given ones located in the folder **TestPowerUp**. They contain code that tests the functionality of your PowerUp class. Replacing source files changes their dates, this required vs to Rebuild instead of just building.

ReBuild (don't just build) and run the application and verify the test steps as described on the console.

Enjoy the result, wipe the spheres with you mouse.

3.2. PowerUpManager

3.2.1. Project

Add a new **project** with name **PowerUpManager** to the **MiniGame** solution, and set it as StartUp project.

Rename the filters and add the **framework** files. Set the props files.

Copy and add your **PowerUp** class files from your previous lab to the Game Files filter. Also copy the image **Resources/Images/PowerUp.png**, don't choose another folder path because the given tests suppose that the image is located in this folder sequence.

Copy and add the given **PowerUpManager** class files to the Game Files filter. This will result in some build errors because the functions aren't defined yet.

3.2.2. General

In this project you complete the given PowerUpManager class. This class manages the PowerUp objects in a game:

- It creates them
- It holds the PowerUp objects in a **std::vector** container class.
- It removes them when a rectangular actor overlaps them.

3.2.3. PowerUpManager class

Add the method definitions. Don't hesitate to add private helper methods to keep your code readable.

The **constructor** and **destructor**.

The **AddItem** method creates a PowerUp object using the given center and type, and adds it to the vector container.

The **Draw** method draws all PowerUp objects that are in the vector.

The **Update** method updates all the PowerUp objects that are in the vector.

The **Size** method returns the current number of PowerUp objects in the vector.

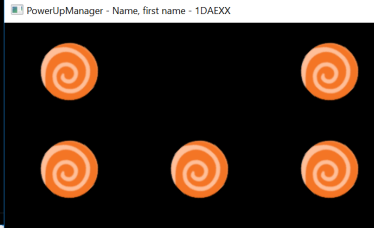
The **HitItem** method removes and deletes the PowerUp object that is hit by the given rectangular shape from the vector (see tip). It returns with true in this case. It removes the first encountered hit Powerup object, thus the remaining powerup objects are not checked.

Tip: to remove an item from the vector, you don't need to move all the following elements in the vector one position towards the beginning of this sequence container. Just copy the last item on this item's position and then remove the last one using the **pop_back** method of the vector.

3.2.4. Test the PowerUpManager class

```
--> Test without drawing <--
--> Create PowerUpManager and verify Size() result <--
--> Call AddItem 20 times and verify Size() result <--
--> Call HitItem 20 times and verify Size() result <--
--> Again call AddItem several times and verify Size() result <--
--> Call destructor <--

--> Init tests with drawing <--
Test Draw, Update and HitItem :
1. Verify that the colored circles coincide with the drawn PowerUp objects.
2. Verify that the PowerUp objects disappear when the mouse's red rectangle hits it.
3. Verify that the number of drawn PowerUp objects is equal to the displayed number.
4. Verify that the game stops without memory leaks, even when still some items weren't hit.
Number of PowerUp objects 12
Number of PowerUp objects 11
Number of PowerUp objects 10
Number of PowerUp objects 9
Number of PowerUp objects 8
```



Overwrite the **Game class** files with the given ones. This given Game class contains some tests of the PowerUpManager class.

Have a look at the Task List of Game.cpp

Task List	
Entire Solution	
Description	
TODO: 1. Uncomment following call of TestWithoutDrawing	
TODO: 2. Uncomment following call of InitTestWithDrawing	

It indicates 2 tests, uncomment them one by one. **If a test fails, then adapt YOUR code and do not change the test code.**

- The first one tests the PowerManager without drawing the PowerUps on the window. It tests the constructor, AddItem, Size and destructor methods. You should build and launch it in Debug configuration. It crashes using assert when a test fails.
- The second test draws on the window and tests the interaction with a rectangular actor. A console message shows what you should verify during this test.

3.3. MiniGame Part 1

1.1.1. Create the project

Add a new project with name **MiniGame** to the MiniGame solution and add the framework files. Set the props files.

Remove and delete the generated file **MiniGame.cpp**.

Adapt the window title and change the window size into **846 x 500**.

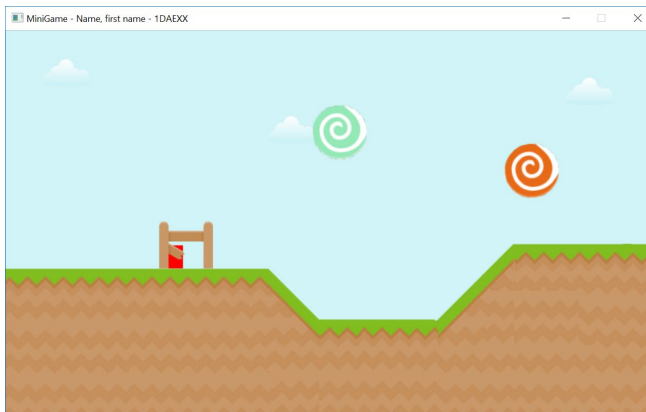
Copy the given **Resources** folder.

Copy and add your **PowerUp** and **PowerUpManager** class files you created in previous parts.

In this project you'll create a small game in which a rectangular avatar moves through a level and undergoes transformations when hitting a power-up.

You'll continue working on it in the coming weeks.

1.1.2. Level class



This class represents the level of a game. It offers this functionality:

- **Collision detection** of a rectangular game actor with the level. The level has a polygonal shape, whose vertices are saved in the data member `m_Vertices`.
- **Drawing** of the background (background.png) and foreground (Fence.png).

Define this class.


a. The UML

This is the UML diagram, decide yourself which methods/data members are const.

Level
- m_Vertices: std::vector<Point2f> - m_pBackgroundTexture: Texture* - m_pFenceTexture: Texture* - m_FenceBottomtLeft: Point2f
+ Level () + DrawBackground (): void + DrawForeground (): void + HandleCollision(actorShape: Rectf&, actorVelocity: Vector2f&): void


```
+ IsOnGround(actorShape: const Rectf&) : bool
```

b. Info about the methods

Constructor
<p>Initializes the data members:</p> <ul style="list-style-type: none"> - both textures (background.png, fence.png), - the position of the fence { 200, 190 } and - the vertices needed for the collision level, these are the values : { 0, 0 }, { 0, 190 }, { 340, 190 }, { 408, 124 }, { 560, 124 }, { 660, 224 }, { 846, 224 }, { 846, 0 }, { 0, 0 }
DrawBackground
Draws the background texture
DrawForeground
Draws the fence texture at the given position
HandleCollision
<p>It handles the vertical collision of the given actor with the level using the Raycast functionality.</p> <p>It stops the actor when it penetrates the level:</p> <ul style="list-style-type: none"> - the bottom-position is changed to the y value of the intersection point (actor should not penetrate the level) and - the vertical part of the actor's velocity becomes 0. <p>Tip: use Raycast with a vertical ray (blue line) in the middle of the actor.</p> 
IsOnGround
<p>Returns true when the actor touches the level, otherwise false is returned.</p> <p>Tip: use RayCast with a vertical ray in the middle of the actor and that is 1 pixel deeper than the bottom.</p>

1.1.3. Test the Level class

Again we give you a helping hand by providing the test code. **Overwrite the Game class** files with the ones in the Minigame/LevelTest folder. This is what this Game class does:

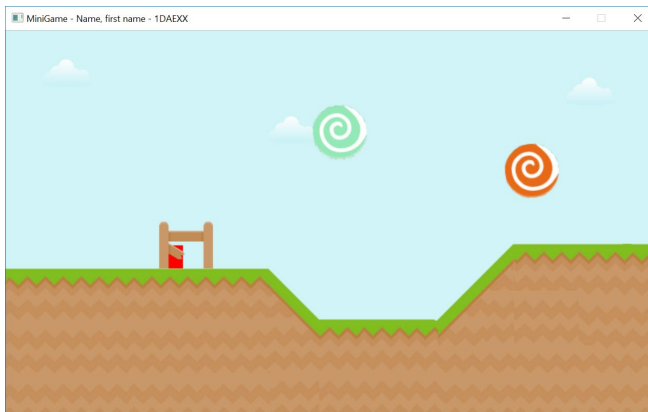
- It creates a PowerUpManager and adds 3 PowerUp objects to it. It draws and updates the PowerUpManager object.
- It also creates a Level object and draws its background and foreground.
- It creates a rectangular actor that undergoes free fall. It's possible to reposition it, just press the LMB and it starts again a free fall at the top of the window and at a horizontal position corresponding with the x position of the mouse.
- The actor stops falling when colliding with the level (HandleCollision is used)
- It gets another color when it is on the ground (IsOnGround is used)

Build, run and verify the items listed on the console.

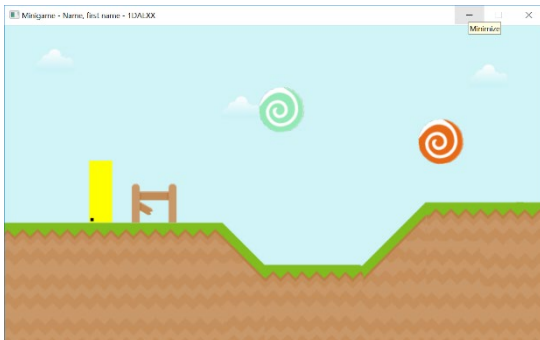
```
--> Level test <--
Verify that:
- The background is drawn ( DrawBackground ).
- The actor stops falling when it collides ( HandleCollision ) with the level.
- The actor is green when it doesn't touch the level and red when it does ( IsOnGround ).
- The actor is drawn behind the fence ( DrawForeground ).
- When the actor hits a power up, this power up disappears.

Clicking with the mouse on the window, repositions the actor at:
- the top of the window
- a horizontal distance corresponding with the mouse click.
```

In this screenshot 1 power-up has been hit, also notice that the actor is drawn behind the fence.



1.1.4. Avatar class



This class defines the avatar of this small game. It is able to perform next actions.

- It **moves** through the level. The avatar movement is controlled using the left/right and up arrow keys. It only listens to these keys when it touches the level (is on ground) and is not transforming.
- It **transforms** when it overlaps one of the power-up items in the game. When this happens, the avatar undergoes a

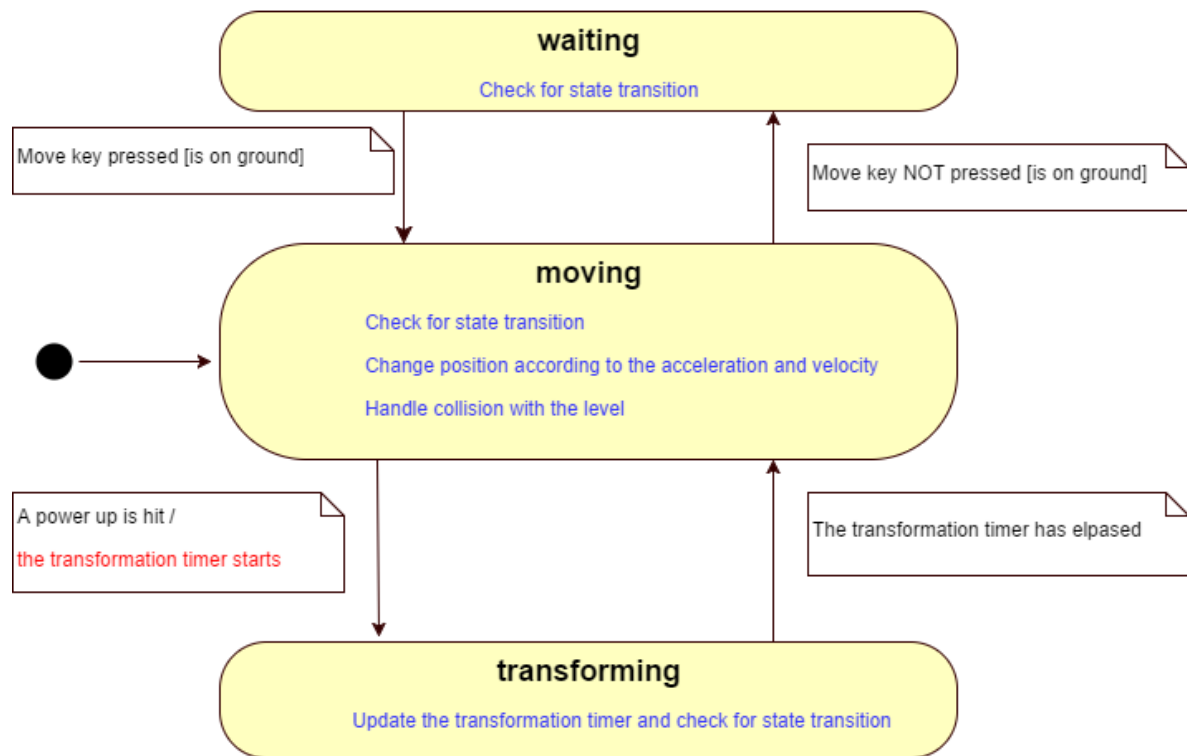
transformation that takes 1 second. During this transformation it doesn't undergo the gravity acceleration nor listens to the movement keys.

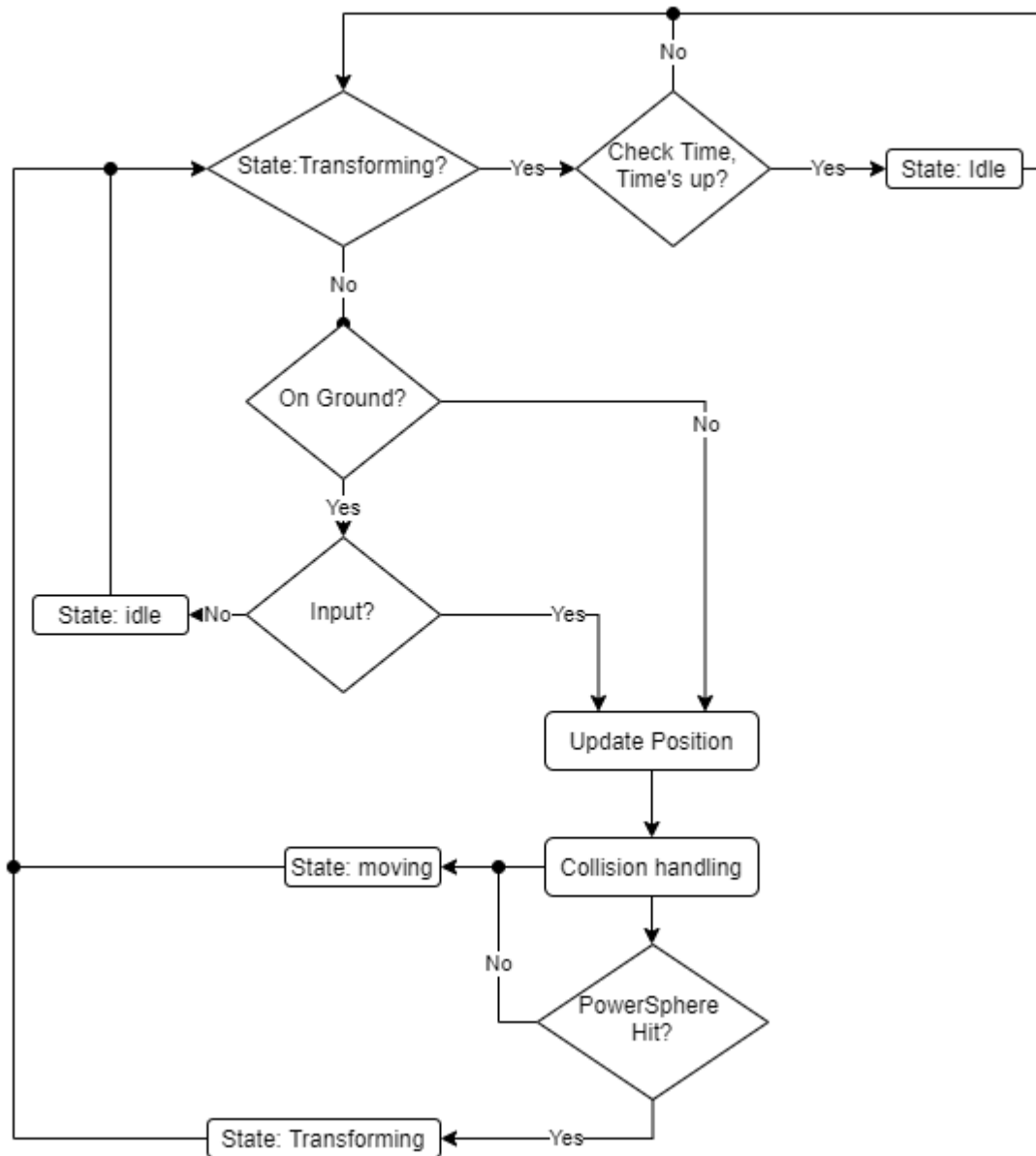
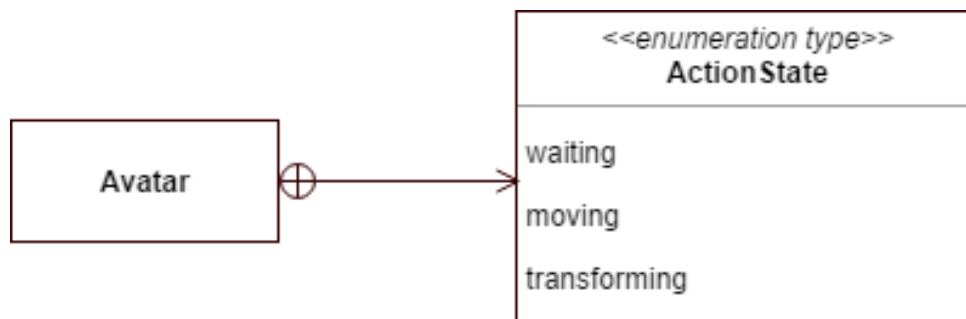
- It just **waits** when none of the above happens.

Thus the avatar can be in 1 of 3 states: **waiting**, **moving** or **transforming**. It also has a power value depending on the number of hit power-ups: 0 to 3.

In a first phase you won't draw an animated avatar, you'll just draw a colored rectangle, the color reflecting its state. And at the bottom a number of filled rectangles indicates the power of the avatar. In the above image, the avatar is waiting and has a power value 1.

a. State diagram



b. Avatar Flowchart**c. Relation Avatar - ActionState****d. Avatar class diagram**

Below class diagram shows only the public functions (not the private ones). However, you're advised to **split up huge functions into smaller helper functions**.

Also, we didn't indicate which methods/data members are const, we expect you to decide about this matter.

Avatar
<ul style="list-style-type: none"> - m_Shape: <code>Rectf</code> = { 50, 280, 36, 97 } - m_HorSpeed: <code>float</code> = 200.0f - m_JumpSpeed: <code>float</code> = 600.0f - m_Velocity: <code>Vector2f</code> = {0.0f, 0.0f } - m_Acceleration: <code>Vector2f</code> = { 0, -981.0f } - m_ActionState: <code>ActionState</code> = <code>ActionState::moving</code> - m_AccuTransformSec: <code>float</code> = 0.0f - m_MaxTransformSec: <code>float</code> = 1.0f - m_Power: <code>int</code> = 0
<ul style="list-style-type: none"> + Avatar() + Update(elapsedSec: <code>float</code>, level: <code>const Level &</code>): <code>void</code> + Draw(): <code>void</code> + PowerUpHit(): <code>void</code> + GetShape() : <code>Rectf</code> - ... your private methods

e. Description of some data members

m_Shape
The rectangular shape that contains the position and dimensions of the avatar
m_ActionState
Indicates the current action state of the avatar
m_AccuTransformSec
The number of seconds the avatar is in the transforming state
m_MaxTransformSec
The number of seconds the transformation should at least last
m_Power
Indicates the number of power-ups the avatar has hit

f. The avatar's velocity

When the left/right arrows are pressed, it gets a horizontal velocity as indicated by the **m_HorSpeed** data member.

When the up arrow key is pressed, it gets an upwards velocity as indicated by **m_JumpSpeed**.

The vertical velocity component is also influenced by the gravity acceleration as indicated by the **m_Acceleration** data member.

g. Description of some methods

PowerUpHit
Tells the avatar that it hit a power-up.

Update
See state diagram

Overwrite the **Game** class files with the ones in the folder

04_Minigame/AvatarTest.

Build, run and verify that the avatar behaves as mentioned on the console.

```
--> Avatar test <--  
Verify that the avatar behaves as follows.  
- Moves along the level when the left/right arrow is pressed.  
- Doesn't move when it is on the ground and no key is pressed.  
- Jumps only when it is on the ground and the up arrow key is pressed.  
- Doesn't move during 1 second when hitting a power up.  
- Starts moving again ( e.g. falling ) after this second.  
- The number of small rectangles in the bottom left corner is equal to the number of hit power ups.  
- Has a red color when it is moving.  
- Has a yellow color when it is waiting.  
- Has a blue color when it is transforming.
```

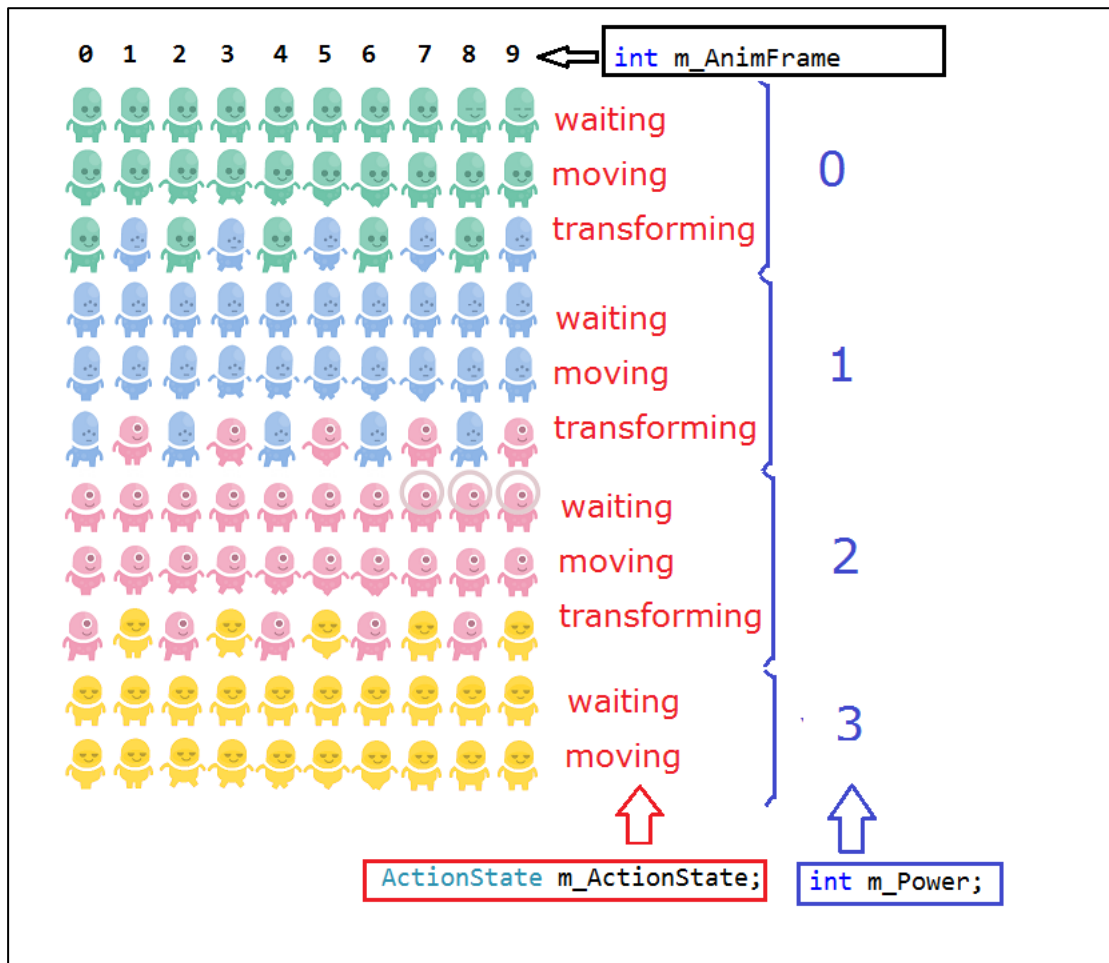
3.4. MiniGame Part 2

In this exercise you'll:

- Add animations to the avatar
- Add a camera,
- Read the level vertices from an Svg file

3.4.1. Animated Avatar class

Now that the basic Avatar works fine, let's add animation using the given sprite sheet. The image below shows that 3 data members - **m_ActionState**, **m_Power** and **m_AnimFrame** - indicate which source rectangle we need to draw.



a. UML class diagram

We need some extra data members to implement the animations. They are mentioned in the UML below, decide yourself which are const



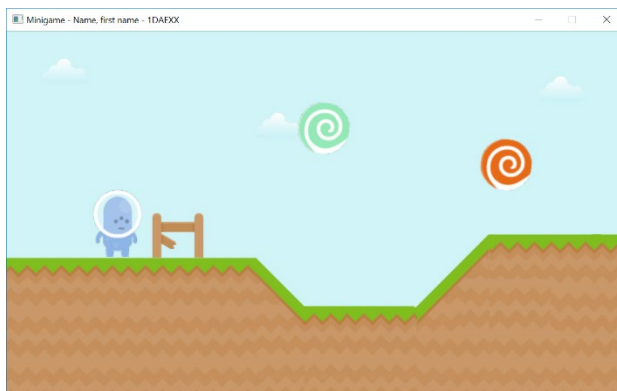
```

---
- m_pSpritesTexture: Texture*
- m_ClipHeight: float = 97.0f
- m_ClipWidth: float = 72.0f
- m_NrOfFrames: int = 10
- m_NrFramesPerSec: int = 10
- m_AnimTime: float
- m_AnimFrame : int
---
- ... your private methods ...

```

b. The new data members

m_pSpritesTexture
A texture pointer for the sprite sheet of the avatar
m_ClipHeight, m_ClipWidth
Indicating the size of a frame in the sprite sheet
m_NrOfFrames
The number of frames (columns) in the sprite sheet. For each animation type there are 10 frames.
m_NrFramesPerSec
Number of frames per second, is for each animation the same: 10 frames per second. Thus each frame lasts $1 / m_NrFramesPerSec$.
m_AnimTime
The accumulated elapsed time in seconds for animation timing (this lets us decide whether it's time to switch to the next animation frame).
m_AnimFrame
Indicating the current frame of the animation.



c. Update method

Count the elapsed time (`m_AnimTime`), and change the current animation frame number (`m_AnimFrame`) when appropriate as indicated by `m_NrFramesPerSec`

Tip: define a helper method that does this task.

d. Draw method

No longer draw a rectangle but use the Texture to draw the avatar.

Build, run and verify that the animation works correctly.

3.4.2. Flip the avatar

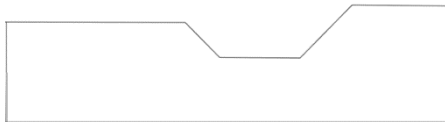
Use the OpenGL model view matrix to draw a flipped avatar when the horizontal velocity is less than 0.

```
glScalef( -1, 1, 1 );
```



3.4.3. Load level vertices from svg file

The level of the mini game is quite simple, it has a limited number of vertices. However for larger levels, there is a way to get the vertices from a level image described by an svg type of file.



This picture is the svg file of our level.

Svg is an xml implementation.

When you open this file with e.g. WordPad you'll notice it contains a **Path element** ([Path element](#)) with a **d attribute** that describes the black outline. Parsing this information leads us to the vertices of the level.

```
height="500"
width="846" />
<path
  style="fill:none;fill-rule:evenodd;stroke:#000000;stroke-
width:1px;stroke-linecap:butt;stroke-linejoin:miter;stroke-
opacity:1"
  d="m -0.8539937,310.4134 343.3054637,0.85399
65.75752,66.61151 152.86487,0.854 99.06327,-100.77126
185.31663,1.70799 L 846,500 0,500 Z"
  id="path3348"
  inkscape:connector-curvature="0" />
</svg>
```

We provide you with the class **SvgParser** which offers a **static** function **GetVerticesFromSvgFile** that parses the content of a given svg file and fills a given vector with the vertices as described by the **d attribute of the path element** in this file.

The Inkscape application offers the possibility to convert an image of your level into an svg file. If you need to that for your game, use the document [HowToCreatSVG.pdf](#) in 00 GameProject / Scalable Vector Graphics.

a. Read vertices from svg file in the Level class

Change the Level class so that it reads the vertices from the given svg file instead of using hard coded vertices values.

Notice that the SvgParser needs a vector of Point2f vectors – it's like a 2d array of Point2f - , this is because there can be more than one path in the svg file. For our svg file this vector will contain only one vector element. The Raycast function expects a vector of Point2f elements, so just pass the first element to the Raycast function.

3.4.4. Camera – part1

Now the window has the same size as the level. When this is not the case a camera that follows the avatar and that doesn't go outside the level boundaries is needed. We'll implement this functionality in a Camera class. In a first phase we will just draw a rectangle that indicates the camera position and size, later we will transform the model view matrix.

a. Define the Camera class

It has the members as indicated in below UML. Decide yourself which members are const.

Camera
<ul style="list-style-type: none"> - m_Width: float - m_Height: float - m_LevelBoundaries: Rectf
<ul style="list-style-type: none"> + Camera(width: float, height: float) + SetLevelBoundaries(levelBoundaries: const Rectf&): void + Draw(target: const Rectf&): void - Track(target: const Rectf&): Point2f - Clamp(bottomLeftPos: Point2f&): void

b. Description of the data members

m_Width
Width of the camera
m_Height
Height of the camera
m_LevelBoundaries
The boundaries that the camera shouldn't leave

c. Description of the methods

Constructor
Initializes the data members. Give the boundaries the initial value: {0, 0, width, height)
SetLevelBoundaries
Changes the m_LevelBoundaries data member
Track
A helper function that positions the camera around the given rectangle <i>target</i> and returns this new position.
Clamp
A helper function that corrects the given camera position so that it doesn't leave the level boundaries.
Draw
Positions the camera in such a way that the given rectangle <i>target</i> is in the center (use Track) and then adjusts this new camera position to the level

boundaries (use Clamp) and draws a blue rectangle at that position with the dimensions of the camera.

d. Extend the Level class

The camera needs to know the level dimensions. So extend the level with a data member (we use caching) and a method to get those boundaries.

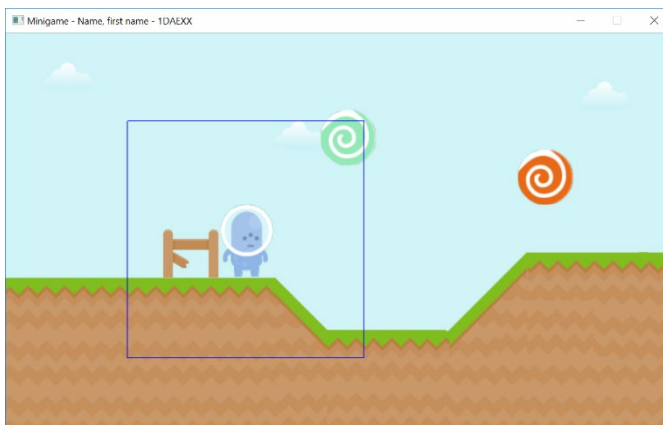
Initialize the boundaries to {0, 0, width of background, height of background}

Level

- m_Boundaries: Rectf

+ GetBoundaries(): Rectf

e. Use the Camera in the Game class



Create a camera object, size is e.g. 300 x 300 and then change the created Camera object its level boundaries to the level's boundaries.

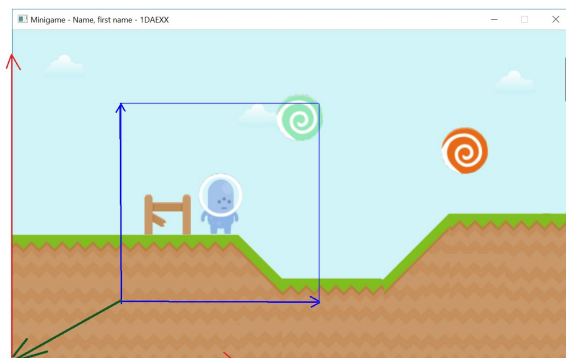
Draw the camera using the avatar's shape to track it.

Build, run and verify that the avatar is surrounded by a blue rectangle that never leaves the level boundaries.

This blue rectangle will be the window in which we'll draw the objects in next phase.

3.4.5. Camera – part2

Now make the window of the game smaller and rename the Draw function into Transform. This function no longer draws the rectangle but transforms (translates) the model view in opposite direction of the calculated camera position.



```
glTranslatef(-cameraPos.x, -cameraPos.y, 0);
```

The following image illustrates this.

In blue: the coordinate axes of the window.

In red: the coordinate axes that we use for the positions of the game objects (world coordinates)

In green: the translation of the blue axes towards the red.

After this translation we actually draw in the **red** coordinate axes.

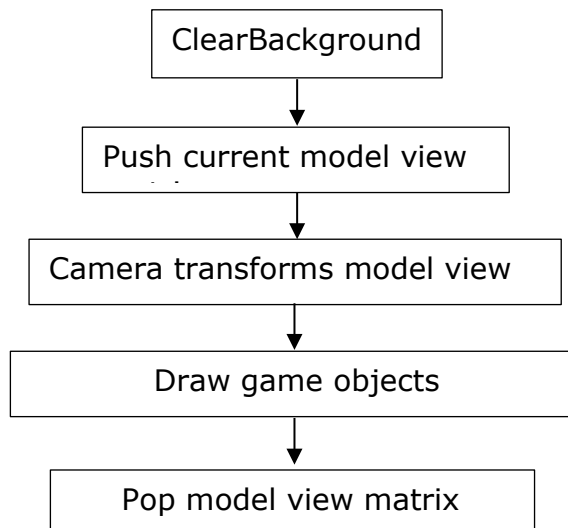
That's why we can keep using the world positions of the objects when drawing them after this translation.

a. Change the window size in main.cpp

Change the size of the game window into e.g. 300 x 250.

b. Change creation of Camera object

Use the window width and height instead of fixed values when creating the Camera object.

c. Change Game::Draw

No longer draw the camera, but before drawing the game objects we call the Transform method (don't forget to push the current model view matrix before calling this function).

And pop this matrix after having drawn all game objects.

Build, run and verify that the camera follows the avatar. Also verify that the camera never leaves the level boundaries.



3.5. MiniGame Part 3



EndSign.png



platform.png

Add the **Resources** folder to the given one, it contains two additional images.

This week you add a platform to the level and detect when the avatar has reached the end of the level. The assignment folder contains a working example.

3.5.1. Platform

Collision with platforms depends on the direction the avatar moves, when moving upwards it shouldn't collide, however downwards it should.

Add a new class **Platform**, that represents a platform in the game. It has the following responsibilities:

- It is able to draw itself

- One can ask to handle the collision of a given shape
- One can ask whether a shape is on the ground of the platform.

a. The UML class diagram

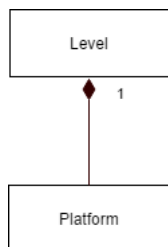
Platform
- m_Shape: Rectf - m_pTexture: Texture*
+ Platform(bottomLeft: const Point2f&): + Draw(): void + HandleCollision(actorShape: Rectf&, actorVelocity: Vector2f&): void + IsOnGround(actorShape: const Rectf&, actorVelocity: const Vector2f&): bool

Decide yourself which methods and data members are const.

b. Description of the methods

Draw
Draws the platform texture
IsOnGround
Returns true when the given actor is not going upwards and is on the platform's top.
HandleCollision
Handles collision with this platform only when the actor is moving downwards.

c. Platform object as part of the Level class

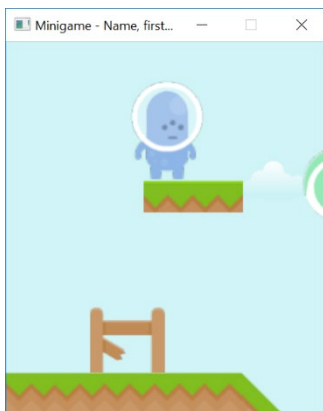


The Level class creates the Platform object and is responsible for the life cycle of it. This is a **composition** relationship.

The Level object draws the platform when drawing its background.

The Level object's collision handling should also verify the collision with the platform. The same goes for the is-on-ground check. However, the Platform needs information about the actor's velocity. Add a velocity parameter to the level's **IsOnGround** method.

Build, run and verify that the avatar is behaving correctly.



- It can jump through the platform
- It lands on the platform when falling.
- It reacts on the move keys when on the platform.

3.5.2. End of the game

Now the avatar can leave the window. Let's draw an end-sign at the end of the level and make the game stop when the avatar reaches this sign.

The level draws the end sign when the foreground is drawn. And it offers functionality that verifies whether an actor reached it.

a. Update the Level class.

Add the data members and functionality as described in this UML. Again, decide yourself whether the new method and data members are const.

Position of the end sign:

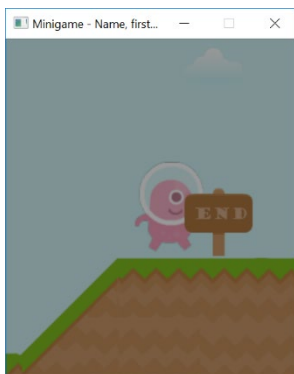
- Left: 730
- Bottom: 224

The **HasReachedEnd** method returns true when the given shape overlaps the end sign shape.

Level
<pre> --- - m_EndSignTexture: Texture - m_EndSignShape: Rectf ---</pre>
<pre> --- + HasReachedEnd(actorShape: const Rectf&) : bool ---</pre>

b. Adapt the Game class

Add a data member **m_EndReached** of type bool which is false at the start.



In the **Update** method, after having updated the Avatar, check whether it reached the end of the level using **Level::HasReachedEnd** and store the result in the new bool data member.

In the Update method, no longer call the update method of the Game objects when the end is reached.

In the **Draw** method, draw a semi-transparent black rectangle above the game objects when the end is reached.

3.6. MiniGame Part 4 - HUD

Add the **Resources** folder by the given one. It contains some images for the HUD.

3.6.1. The HUD

The HUD represents in a graphical form the number of power ups the avatar has hit. Add a class with name HUD and define it as defined in the paragraphs below. Decide yourself which data members/methods are const.

a. UML class diagram

HUD
- m_BottomLeft: Point2f - m_TotalPowerUps: int - m_HitPowerUps: int - m_pLeftTexture: Texture* - m_pRightTexture: Texture* - m_pPowerUpTexture: Texture *
+ Hud(topLeft: const Point2f&, totalPowerUps: int) + Draw(): void + PowerUpHit():void

b. Description of the data members

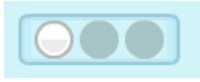
m_BottomLeft
Position of the bottom left corner of the HUD on the window
m_TotalPowerUps
The total number of power ups in the game
m_HitPowerUps
The number of hit power ups
m_LeftTexture, m_RightTexture, m_PowerUpTexture
Texture objects for:
<ul style="list-style-type: none"> The left side of the HUD (HudLeft.png) The right side of the HUD (HudRight.png) A power up (hit/not hit) in the HUD (HudPowerUp.png)

c. Description of the methods

Constructor
Initializes the data members
PowerUpHit
Indicates that a power up has been hit, it increments the related counter.
Draw
Draws the HUD using the given Texture objects.

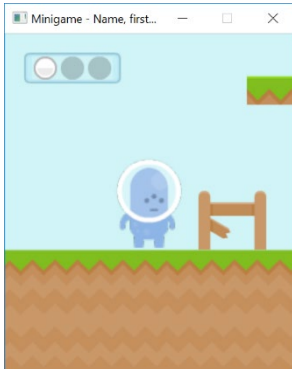
The number of power ups in the HUD is the total number of power ups

The hit ones get a white color, the remaining ones a greyish color, e.g. the screenshot below indicates that there are 3 power ups and that one is hit by the avatar.



d. Use it in the Game class

Make an instance of the HUD class



3.7. MiniGame Part 5 - Sound

3.7.1. Introduction exercise

In this exercise you're going to play sound effects and music using the SDL extension library `SDL_mixer` ([SDL mixer tutorial](#)).

a. Create the project

Add a new project with name **Sound** to the weekly solution and delete the generated file **Sound.cpp**.

Use the framework.

The framework contains 2 classes for sound functionality: `SoundEffect` and `SoundStream`. The SDL mixer is used. It is opened/closed in the **Initialize** / **CleanUp** method of the `Core` class.

Overwrite the `Game` class files by the given ones in `sound.zip`.

Also copy the given **Resources** folder.

b. The `SoundStream` class

This class can be used to play music for an extended period of time. It uses the struct **Mix_Music** defined in `SDL_Mixer.h`.

`SDL_Mixer` offers several functions that work with this struct ([Functions working with Mix Music music](#)), some of them are used in the given `SoundStream` class.

The constructor loads the music from a given file using the [Mix_LoadMUS](#) function, several types can be loaded (wav, mp3, ...). With the function **IsLoaded** one can check whether the music file was loaded successfully. **Play** starts playing the music.

You can play only **one sound stream at a time**. You can use this class for the background music of a game.

c. `SoundEffect` class

This class can be used to play sound effects. It uses the **Mix_Chunk** struct defined in `SDL_Mixer.h`.

`SDL_Mixer` offers several functions that work with this struct ([Functions working with Mix Chunk samples](#)), some of them are used in the given `SoundEffect` class.

The constructor loads the sample from a given file using the [Mix_LoadWAV](#) function, the name is misleading, it not only supports wav files but also other types (ogg, mp3, ...). With the function **IsLoaded** one can verify whether the sample was loaded successfully. The function **Play** plays the chunk.

You can play multiple samples at once. You can use this class for sound effects in your game such as firing, power ups,...

d. Use the Sound classes in the `Game` class

Complete the given `Game` class with sound functionality, just follow the TODO list, that guides you in :

- The creation of the necessary `SoundStream`/`SoundEffect` objects.
- The handling of the user requests.
- The stopping to the sound(s) when the user switches between the 2 test series.

TODO: 1. Create the DonkeyKong and Mario SoundStream objects

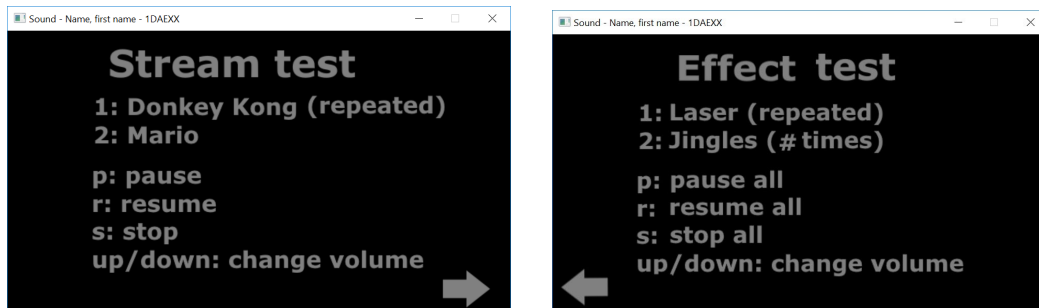
TODO: 4. Create the Laser and Jingles SoundEffect objects

TODO: 3. User switches to the "Effect test", stop the music

TODO: 6. User switches to the "Stream test", stop the effects

TODO: 2. Handle the user requests in the "Stream test" menu

TODO: 5. Handle the user requests in the "Effect test" menu



3.7.2. Power up hit sound

Copy the given Resources\Sounds folder in your Resources folder. It contains an mp3 file with the sound to be played when the avatar hits a powerup.

Change the **PowerUpManager** class. Play the given powerUp.mp3 when a power up is hit. Don't create the SoundEffect object each time it needs to be played.

4. Submission instructions

Upload the compressed folder *1DAExx_MinGame_name_firstname together with your weekly solution*. It should contain 3 solutions: The weekly solution, e.g. W03, the minigame and your game.

Don't forget to clean the solutions before adding them to the zip file:

- Delete .vs folder
- Delete all the nested debug and x64 folders