

Vector & Transformations

1. Content

| | |
|--|----|
| Vector & Transformations..... | 1 |
| 1. Content | 1 |
| 2. Objective | 2 |
| 3. Exercises | 2 |
| 3.1. New Framework using Classes..... | 2 |
| 3.1.1. One solution having many projects..... | 3 |
| 3.1.2. Create the project..... | 3 |
| 3.1.3. General structure..... | 3 |
| 3.1.4. Detecting heap memory problems | 4 |
| 3.2. Introduction | 5 |
| 3.2.1. Ball class..... | 5 |
| 3.2.2. The Texture class | 6 |
| 3.2.3. Create and draw some Texture objects..... | 6 |
| 3.3. VectorContainerBasics | 6 |
| 3.3.1. Create the project..... | 6 |
| 3.3.2. General..... | 6 |
| 3.3.3. Vector of integer numbers..... | 7 |
| 3.3.4. Vector of Card object pointers..... | 8 |
| 3.4. TrafficLights | 9 |
| 3.4.1. Create the project..... | 9 |
| 3.4.2. General..... | 10 |
| 3.4.3. State machine..... | 10 |
| 3.4.4. State diagram | 10 |
| 3.4.5. TrafficLight class..... | 11 |
| 3.4.6. Create TrafficLight objects in the Game class | 13 |
| 3.5. MiniGame - PowerUps | 14 |
| 3.6. RotatingCards..... | 14 |
| 3.6.1. Create the project..... | 14 |
| 3.6.2. General..... | 15 |
| 3.6.3. Adapt the Card class | 15 |
| 3.6.4. Create Card objects | 15 |
| 4. Submission instructions | 15 |
| 5. References | 16 |

| | | |
|--------|--|----|
| 5.1. | The std::vector<element_type> sequence container | 16 |
| 5.1.1. | Vector container | 16 |
| 5.1.2. | Instantiating a vector | 16 |
| 5.1.3. | Number of elements in a vector – size method | 16 |
| 5.1.4. | Get internal array – data method | 16 |
| 5.1.5. | Access an element – operator[] | 16 |
| 5.1.6. | Adding an element at the end – push_back method | 16 |
| 5.1.7. | Removing an element at the end - pop_back method | 16 |
| 5.1.8. | Range-based for loop | 16 |
| 5.2. | State machine | 16 |
| 5.2.1. | Finite state machine | 16 |
| 5.2.2. | State machine UML | 16 |

2. Objective

At the end of these exercises you should:

- Be able to define and use the basic functionality of the **vector** sequence container from the standard library.
- Know what a **state machine** is and understand a **state diagram**.
- Know how to indicate **transformations** (translations, rotations or scaling) for OpenGL objects in the OpenGL **model matrix**.
- Know how to use the given **Matrix2x3** functionality to transform vertices.

Also you'll rehearse:

- Defining classes
- Creating objects of classes
- Use the object's services (methods)

We advise you to **make your own summary of topics** that are new to you.

3. Exercises

Your name, first name and group should be mentioned at the top of each cpp file.

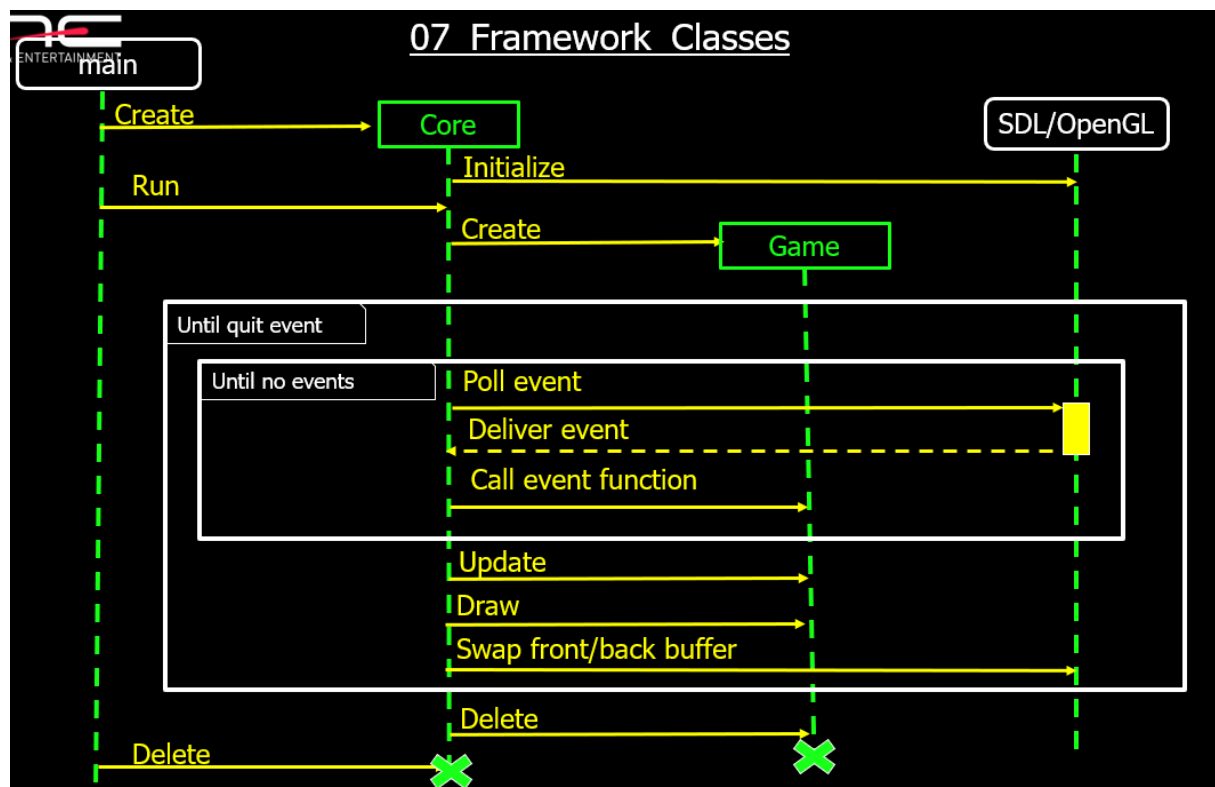
Give your **projects** the same **name** as mentioned in the title of the exercise, e.g. **VectorContainerBasics**. Other names will be rejected.

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

3.1. New Framework using Classes

Now that you know how to define and instantiate classes, we will split up the SDL and OpenGL framework code into 2 new classes named **Core** and **Game**. These

classes are given, the objective of this exercise is to understand how to use them in your following game projects.



3.1.1. One solution having many projects

In this course programming2 you will group all the exercises of one week in one Visual Studio solution with name Wxx (xx being the week number).

To do this, follow the instructions in 00_General / ManualsAndGuides / HowToCreateAMultiProjectSolution.pdf

Create a solution with name W01 in your **1DAExx_01_name_firstname** folder.

3.1.2. Create the project

Add a new SDL class project to the solution with name **Introduction**.

Delete the generated Introduction.cpp file.

Use the **classes** framework. Hereby allow to overwrite the created **pch.h** file by the one provided by the framework.

Adapt the window title in main.cpp.

Build and run, you should get a window with your name in the window title.

3.1.3. General structure

a. The Core class

The Core class contains the data members and functionality that was grouped in the **coreDeclarations** and **coreImplementations** regions. And as you know, it offers following functionality:

- All general initialization: SDL, window creation, OpenGL Context, PNG loading, SDL_ttf initialization, ...

- **Creation of the Game object**
- Handling events in the event loop and **informing this Game object about the events** by calling its event functions.
- Time measuring and calling this **Game** object 's **Update** and **Draw method** every frame

b. The Game class

The Game class contains the data members and functionality that was grouped in the **gameDeclarations** and **gameImplementations** regions.

It manages (creates, updates, draws, deletes...) all your game objects such as: Avatar, Enemies, Power ups... and handles the events.

c. The Window struct

The Window struct contains the variables of the **windowInformation** region.

Both The Core and Game class need information about the **window**, such as the title and the dimensions

- The Core class: because it needs this information when it creates the SDL window
- The Game class: because it needs to know the dimensions

That's why we grouped the window properties in a **Window** struct. We will instantiate it in the main function and **pass it to the Core object** when creating it. The Core class will **pass it to the Game object** when creating it.

d. The main function

In the main function a Core object is created, and its Run method is called. The file main.cpp is given.

e. Precompiled headers.

You can precompile headers that include files that hardly ever change and are frequently used in your project. This will reduce the build time and you no longer have to explicitly include them in your code files.

All you need to do is adding a pair of special files (called pch.h and cpp) to your project and add these include directives in pch.h. Then you add **#include "pch.h"** directive as first line to each cpp file of your project.

In previous Visual Studio versions these files had another name: stdafx.h/cpp

[Precompiled headers](#)

Both the Core and Game class and later also some of your own classes use SDL and OpenGL. As a consequence, you need to add SDL and OpenGL related directives in each of these cpp files. That's why we centralize these directives in the file pch.h of the framework. This file is given.

3.1.4. Detecting heap memory problems

Notice that a function **StartHeapControl** is called in the main function. This code reports heap corruption and memory leaks in the output window.

Let's see how these problems are reported. Create in the main function - after the StartHeapControl function call - a dynamic array of some integers without freeing this memory at the end.

When you stop the application the output window reports a memory leak.

```
Detected memory leaks!  
Dumping objects ->  
{154} normal block at 0x005E2800, 120 bytes long.  
Data: < > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD  
Object dump complete.
```

Now free the memory - before the **DumpMemoryLeaks** function call - and test again. The output window should no longer report any memory leaks.

3.2. Introduction

3.2.1. Ball class

Let's see how we can manage the objects that are part of our game in the Game class. We'll make objects of the given Ball class which holds the definition of a bouncing ball.

Copy the given (codefiles.zip) Ball class files in the folder of this project and add them to this project in Visual Studio.

Game is now a class too. Declare an nullptr initialized array of two Ball pointers in Game.h, as private(!) members. Forward declare the Ball class above the Game class declaration. The size of the array is now defined by a **static** const int. That is because the size is no longer a global variable. It will be explained in the theory class this week.

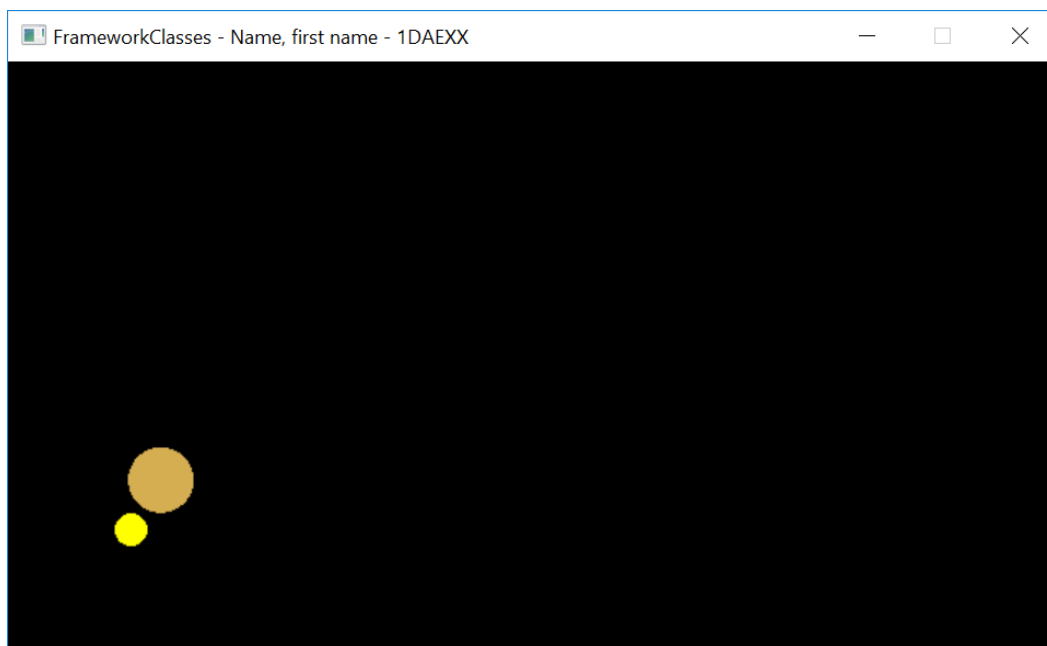
In the Initialize member function, that replaces the Start function, make 2 instances of the Ball class and store their pointers in the array.

Update and draw them.

Delete them too.

Have a look at the report of the memory leak when you don't delete the Ball object created on the heap.

Memory leaks in your projects result in a lower grade.



3.2.2. The Texture class

In previous labs, we already created and drew textures from images and from a string rendered in a specified font. At that time, we only had functions to work with. We **needed** a **struct** to store texture related data such as the dimensions and the texture id.

Now that we know **classes**, we can make a class that contains next to this **data** also the **methods** to **create** and **draw** the **texture** (encapsulation).

Have a look at the given **Texture class**. Notice that it contains the data members and functionality that was grouped in the **textureDeclarations** and **textureImplementations** regions.

3.2.3. Create and draw some Texture objects

Create 2 Texture objects on the heap, one for the DAE image and the other for some text. The texture pointer is declared in the header file of the game class. In the definition of the Initialize member function, the texture object is created. In the Cleanup member function, the texture object is deleted.

Then draw the text-texture 3 times on the window.

Also draw the DAE-texture 3 times using a destination rectangle with decreasing width.

Also test some possible error cases, verify what happens when a wrong file was specified (wrong path).

3.3. VectorContainerBasics

3.3.1. Create the project

Create a new **classes framework solution** with name **VectorContainerBasics**.

3.3.2. General

In this project you'll learn how to use the **std::vector** container which is a sequence container ([Vector container](#)) that is part of the C++-standard library. There are many more container types that will be discussed at the end of this course. For now, we introduce here the very popular **std::vector**.

An **std::vector** contains a number of elements that automatically expands, you can get a pointer to the first element in this array ([Get internal array – data method](#)). You may need this pointer e.g. when you want to call an existing function that expects a parameter of type pointer to the first element instead of an **std::vector**.

To become acquainted with this container, let's make some basic exercises using:

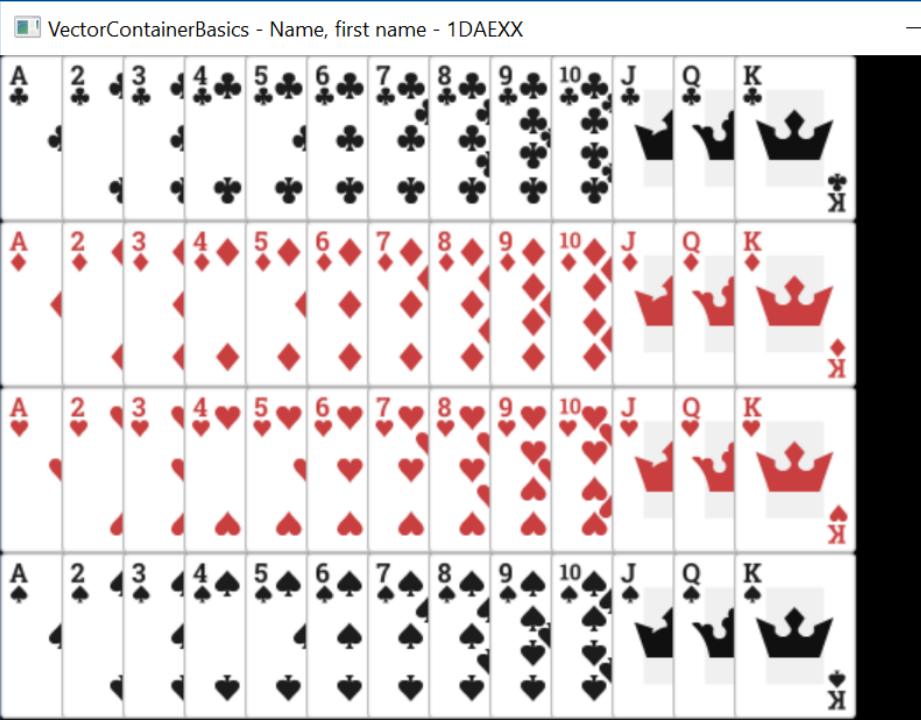
1. a vector of int type elements and
2. a vector of pointer type elements.

At the end of this exercise, you should have a console and window that looks like this.

```

+: Add a number at the end of the vector
-: Remove last number from the vector
^: Increment all numbers in the vector
v: Decrement all numbers in the vector
h: Show menu
25
25 30
25 30 25
25 30 25 29
25 30 25 29 18
25 30 25 29 18 2
25 30 25 29 18
25 30 25 29
25 30 25
26 31 26
27 32 27
28 33 28
29 34 29
28 33 28
27 32 27
26 31 26

```



3.3.3. Vector of integer numbers

a. Adding and removing elements

Define in Game.h a vector of **int** type numbers ([Instantiating a vector](#)).

When the **+** key is pressed, you add an integer with a random value in the interval [0,30] to the vector ([Adding an element at the end – push_back method](#)).

```

5
5 8
5 8 16
5 8 16 6
5 8 16 6 5
5 8 16 6 5 26

```

When the **-** key is pressed, you remove the last element from the vector ([Removing an element at the end – pop_back method](#)).

```

5 8 16 6 5 26
5 8 16 6 5
5 8 16 6

```

After each change, print the elements of the vector as explained in the following section.

b. Accessing elements using the operator[] (subscript operator)

Define a method – **PrintElements** – that prints the elements of the vector to the console.

Use an indexed loop: the vector's **size** method returns the number of elements ([Number of elements in a vector – size method](#)) and with the **operator[]** you get the element at the specified index ([Access an element – operator\[\]](#))

Call this PrintElements method after each change to the vector.

c. Ranged based for loop

Comment the code in this PrintElements method and print the elements using a ranged based for loop, this is without using an index ([Range-based for loop](#))

d. Change the elements

When the **up-arrow key** is pressed, you increment each element with the value 1 and print them.

```
5 8 16 6
6 9 17 7
7 10 18 8
8 11 19 9
```

When the **down-arrow key** is pressed, you subtract the value 1 from each element and print them.

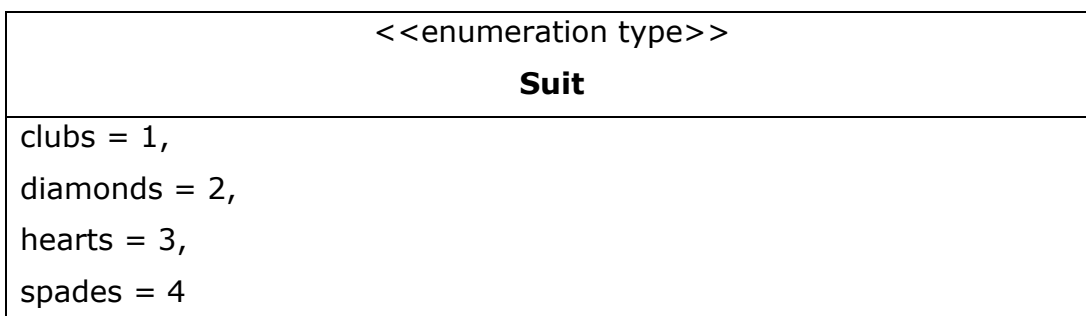
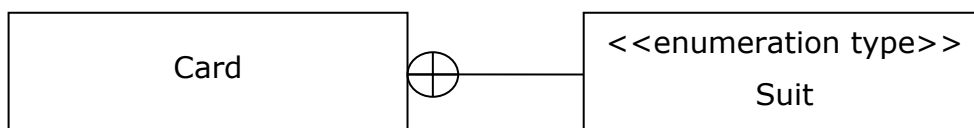
```
8 11 19 9
7 10 18 8
6 9 17 7
5 8 16 6
4 7 15 5
3 6 14 4
2 5 13 3
```

Try changing the elements using both loops (index – ranged-based), what do you notice ?

3.3.4. Vector of Card object pointers

Copy the given Card.cpp and h files in the folder of this project and add them to the project.

a. Define the Card class




```

- m_pTexture: const Texture*
- m_Suit: const Suit
- m_Rank: const int
- minRank: const int
- maxRank: const int
- width: const float
- height: const float

+ Card(suit: Suit, rank: int)
+ ~Card()
+ Draw(destRect: const Rectf& ): void {query}
+GetWidth():float {query}
+GetHeight():float {query}
- GetImagePath(suit: Suit, rank: int ): std::string {query}

```

The given Card class is partly defined, complete it.

The Card images are available in the given Resources folder. The image files have a name that follows a convention: suit number * 100 + rank number. To get the Suit number just cast the Suit enumeration value to an integer.

b. Vector of pointers to Card objects in Game class

Let's create Card objects on the heap and store the pointers in an `std::vector`.

Create Card objects of the 4 suites, for rank numbers in the inclusive interval `[Card::minRank, Card::maxRank]`.

Draw them on the window (in 4 rows), scale factor is 0.5.

When the **s key** is pressed, shuffle the cards.

Notice that the Card objects are not deleted automatically when the vector goes out of scope. Solve this because memory leaks are not what we want.



3.4. TrafficLights

3.4.1. Create the project

Create a new **framework solution** with name **TrafficLights**.

Adapt the window title.

3.4.2. General

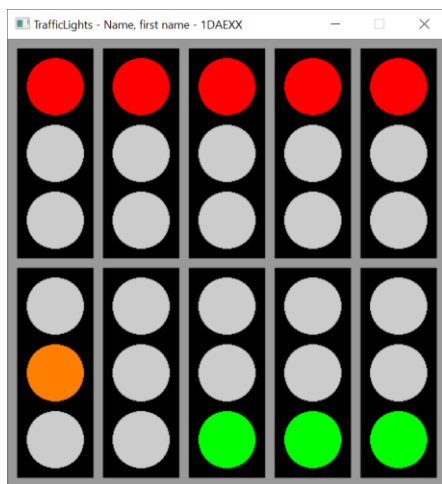
In this project you use a **state machine** which you'll definitely need when programming games. You create a class that represents a traffic light. At start the traffic light is out of service.

Clicking with the LMB on one of its three lights, puts the traffic light in service: the hit light is switched on and the time measuring of that light starts. When the time of that light elapses, the next light is switched on.

Clicking on the armature of the traffic light, switches the traffic light off again.

The **utils namespace** of the framework contains functionality to verify whether a point is in a shape (circle, rectangle, polygon).

At the end of this exercise you should have a window that looks like this. But before starting to code, read in following sections how to proceed.



3.4.3. State machine

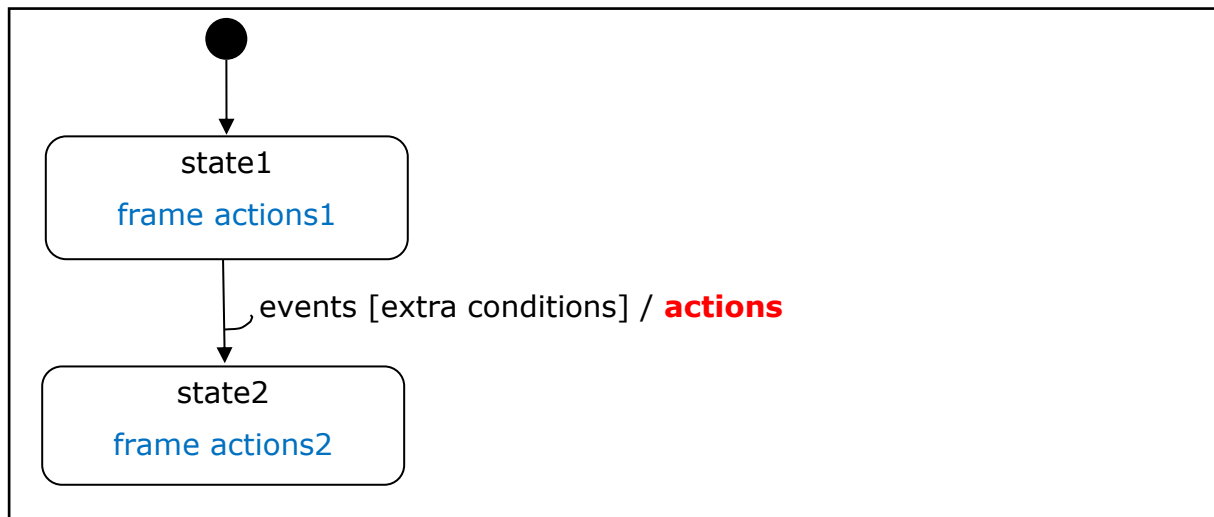
A traffic light can be in one out of **4 states**: red, orange, green or switched off. It is not able to switch immediately from any state to any other, for example, when it is orange it cannot switch straight to green. The options for the next state that an object can enter from its current state are referred to as **state transitions**.

The set of states, the set of transitions and the variable to remember the current state form a **state machine** ([Finite state machine](#)).

3.4.4. State diagram

The states and transitions of a state machine can be represented using a graph diagram or **state diagram** ([State machine UML](#)), where the nodes represent the **states** and the arrows between the nodes represent the **transitions**. The state of an object can only go to another state along one of the arrows.

Next picture shows the general form of a state diagram with 2 states.



It depicts:

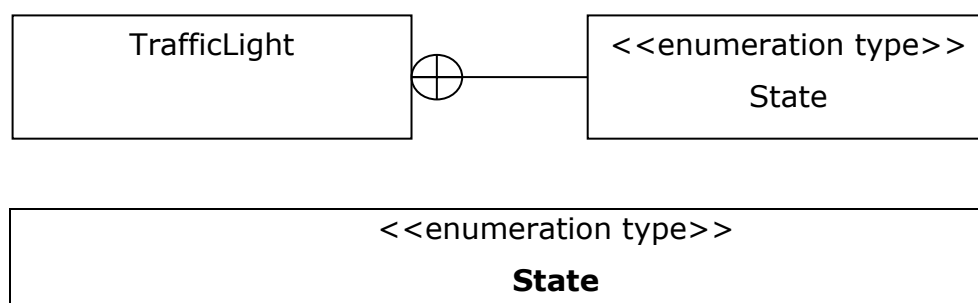
- The possible states of an object as **nodes** with the name of the state, In the above pictures the possible states are: *state1* and *state2*.
- Which state the object gets at **creation**, indicated with an arrow having a filled circle at one side and pointing to the node with the starting state. The starting state is *state1* in the above picture.
- **Transitions**: being in a given state, transitions indicate **into which other states** the object can go. They are indicated with an arrow going from one node to another. In the above picture a transition is possible from *state1* to *state2* but not from *state2* to *state1*.
- Next to the state transitions arrows, we indicate which event and extra conditions cause this state transition of the object, in a predefined sequence and format: **event [extra conditions] / actions**
 1. **Event**: the that triggers this state transition,
 2. **extra conditions**: conditions that have to be met, to make the transition happen
 3. **actions**: actions that should be performed when this state transition happens
- In the nodes we 'll also indicate in blue which actions have to be executed each frame while being in this state.

Let's use such a state diagram to describe the states and state transitions of the traffic light.

3.4.5. TrafficLight class

Add a class TrafficLight to the project. The class defines an enumeration type **State** with 4 possible named values: red, green, orange or off.

a. UML class diagram



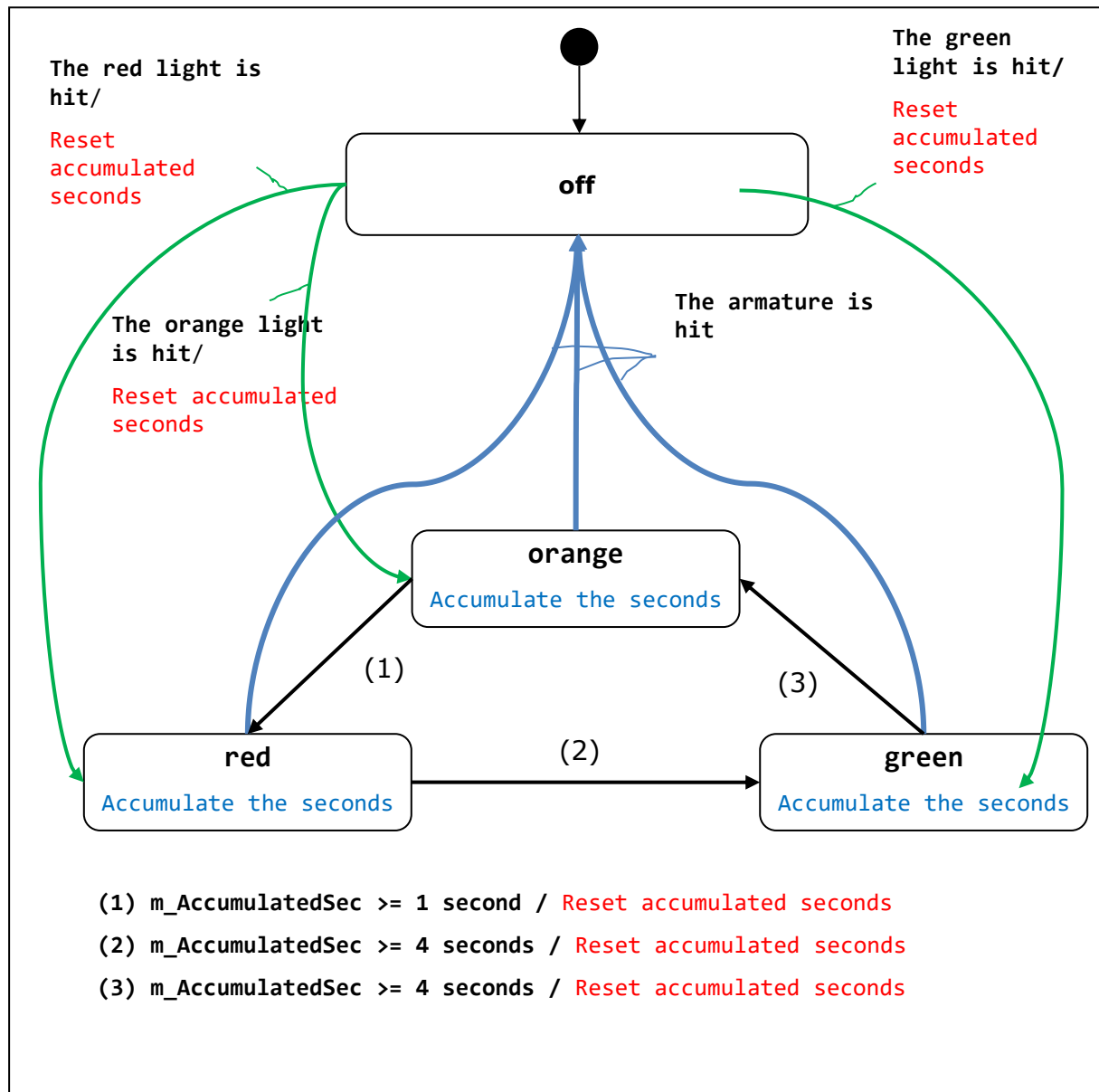
```
green
orange
red
off
```

| TrafficLight |
|--|
| <pre>- m_State: State - m_Position: Point2f - m_AccumulatedSec: float ---</pre> |
| <pre>+ TrafficLight (pos: const Point2f&) + DoHitTest(point: const Point2f&) + Update(elapsedSec: float) + Draw() + GetWidth(): float + GetHeight(): float ---</pre> |

This class diagram doesn't indicate which parameters/methods should be const, that's up to you. Also don't hesitate to add private data/function members, that's what the sequence "---" stands for.

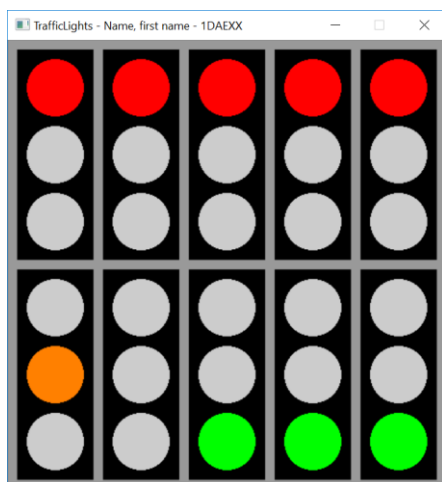
b. State diagram

Following state diagram describes the states and state transitions of the traffic light. Use it to define the methods.

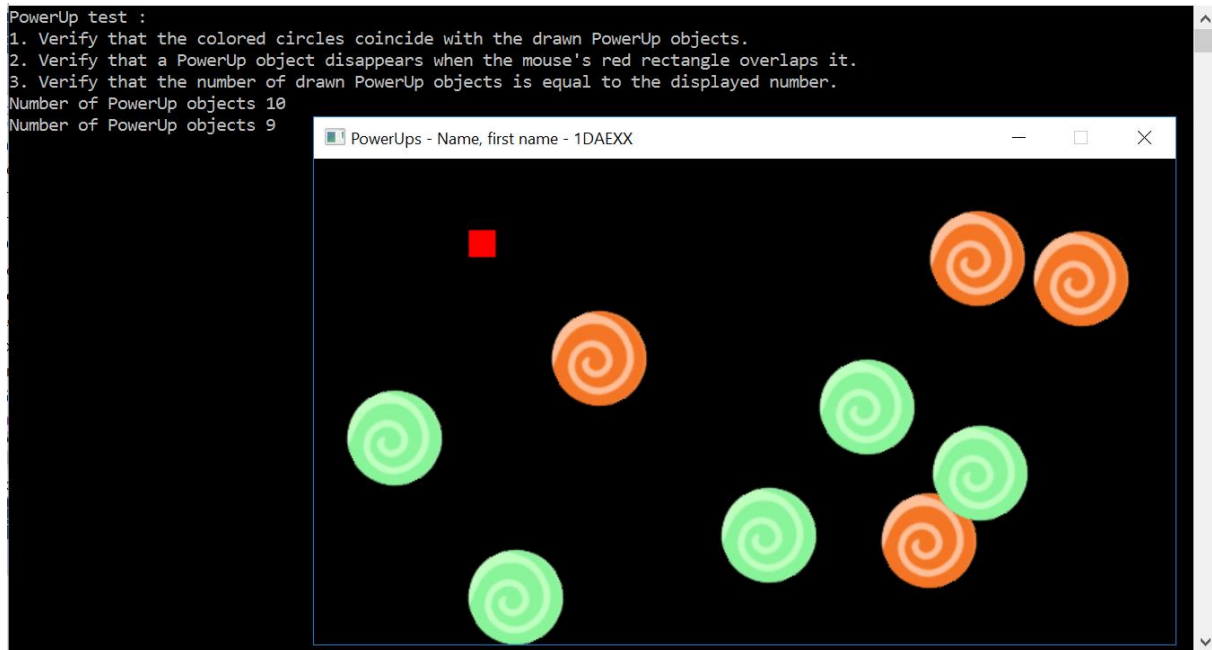


3.4.6. Create TrafficLight objects in the Game class

Create 10 TrafficLight objects using dynamic memory allocation, positioning them in 2 rows. Store the pointers in a `std::vector`. Update and draw them. With the LMB the user can turn a traffic light on or off. Check for memory leaks.



3.5. MiniGame - PowerUps

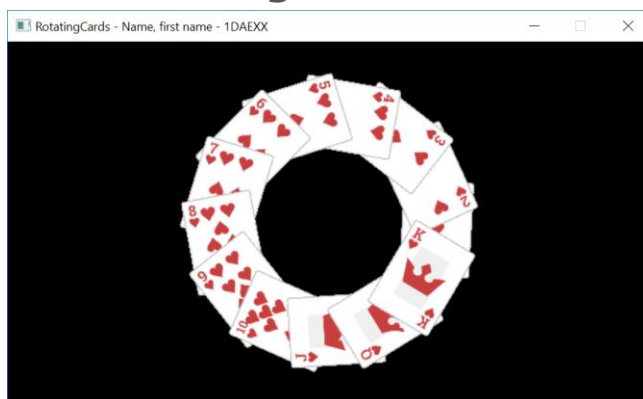


To know how to rotate textures, wait for the demo or watch the video.

Open the **MiniGame** document in 00_MiniGame and complete the Powerup Part.

Don't skip this because the MiniGame series builds up over several lessons. The powerup class is one game object – pickup.

3.6. RotatingCards



3.6.1. Create the project

Add a new **framework project** with name **RotatingCards** to the **W02** solution, and set it as StartUp project.

Delete the generated file **RotatingCards.cpp**.

Rename the filters into "**Framework Files**" and "**Game Files**".

Copy and add the framework files.

Copy and add the Card class files from a previous exercise.

Adapt the window title.

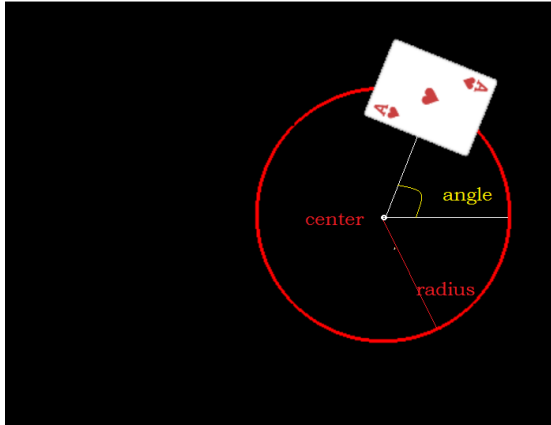
3.6.2. General

In this project you'll create Card objects of one suit and make them rotate round a point in the middle of the window using the model matrix transformations of OpenGL.

3.6.3. Adapt the Card class

The Card class should know the rotating circle (center and radius) and the starting angle, this leads us to some extra data members.

RotatingCards - Name, first name - 1DAEXX



It needs an Update method to change its angle (rotating speed is e.g., 1 complete rotation per second):

```
m_Angle += m_RotationalSpeed * elapsedSec;
```

The Draw method no longer needs the position, because the Card knows the rotating circle and the angle.

```
Card( Suit suit, int rank, float angle, const Circlef& circle );
void Draw( ) const;
void Update( float elapsedSec );
```

3.6.4. Create Card objects

Create the Card objects of one suit, giving each one of them another starting angle, so that they all get their position on the rotating circle, like this:



Update and Draw them.

4. Submission instructions

You have to upload the folder *1DAExx_01_name_firstname*, which contains the *W01 solution* and the *Minigame Solution* hereby following these steps:

- Close the W01 Visual Studio solution.
- Remove all the folders that we don't need and take a lot of space: **Debug, x64 and .vs**. They are not only in the solution folder, but also in each project folder!
- Compress this *1DAExx_01_name_firstname* folder and upload it before the start of the first lab next week.

5. References

5.1. The `std::vector<element_type>` sequence container

5.1.1. Vector container

<http://www.cplusplus.com/reference/vector/vector/>

5.1.2. Instantiating a vector

<http://www.cplusplus.com/reference/vector/vector/vector/>

5.1.3. Number of elements in a vector – size method

<http://www.cplusplus.com/reference/vector/vector/size/>

5.1.4. Get internal array – data method

<http://www.cplusplus.com/reference/vector/vector/data/>

5.1.5. Access an element – operator[]

[http://www.cplusplus.com/reference/vector/vector/operator\[\]/](http://www.cplusplus.com/reference/vector/vector/operator[]/)

5.1.6. Adding an element at the end – push_back method

http://www.cplusplus.com/reference/vector/vector/push_back/

5.1.7. Removing an element at the end - pop_back method

http://www.cplusplus.com/reference/vector/vector/pop_back/

5.1.8. Range-based for loop

<http://www.cprogramming.com/c++11/c++11-ranged-for-loop.html>

5.2. State machine

5.2.1. Finite state machine

https://en.wikipedia.org/wiki/Finite-state_machine

5.2.2. State machine UML

https://en.wikipedia.org/wiki/UML_state_machine