# Operator overloading

## 1. Content

# 2. Objective

At the end of these exercises, you should be able to:

- Implement operator overloading in classes.
- Decide on overloading using member or non-member function
- Transform positions using matrix transformations
- Use the Vector2f structure to make diffuse lighting calculations.

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 as comment at the top of each cpp file.

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

## 3.1. OperatorOverloadingBasics

### 3.1.1. Create the project

Add a new console project with name **OperatorOverloadingBasics to your W05 solution** in your **1DAExx_05_name_firstname** folder.

Copy the given file OperatorOverloadingBasics.cpp in the folder of this project (overwrite the existing one).

Add your name and group as comment at the top of the cpp file.

Building and running this project shouldn't give any problems.

### 3.1.2. Test functionality is given

Have a look at the content of the given file **OperatorOverloadingBasics.cpp.** Notice it contains a lot of commented code (include statements, function declaration, calling tests functions, body of the test functions). This is code that tests the correct working of the operators that you are going to define as described further in this document. These operators and classes are not part of the project yet, so that's why they are still commented otherwise this code would result in errors.

You'll uncomment the tests step-by-step as you proceed.

### 3.1.3. Health class with binary arithmetic operators

#### a. Add the given Health class

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

Have a look at this class. It's a very simple class, it manages the health of a game entity.

#### b. Add binary arithmetic operators to the Health class

Binary arithmetic operators

These operators should make it possible to use the + and - operator to add/ subtract an integer health value to/from a Health-object. It is a common and good practice to first define the compound addition operator (**Health += int)** and then to call this one in all the following addition/subtraction operator overloading functions.

Your operators should have the expected behavior, e.g. nr1 += nr2 not only adds nr2 to nr1 but also has a result: the resulting nr1 value. That's why the **Health += int** operator not only adds the integer value to the Health object but also should return the **Health&** object that contains the addition result.

```
int nr1{ 2 };
int nr2{ 3 };
int nr3{};
nr1 += nr2; // nr1 becomes 5
nr3 = nr1 += nr2; // nr3 becomes 8
```

What would be the content of nr1, nr2 and nr3 when we add after previous code snippet next line of code?

```
nr3 = nr1 += nr2 += nr1;
```

Good practices from the book "**Programming: Principles and Practice Using C++ (2nd Edition) By Bjarne Stroustrup**"

*It is generally a good idea not to define operators for a type unless you are really certain that it makes a big positive change to your code.*

*Also, define operators only **with their conventional meaning**: `+` should be addition, binary `*` multiplication, `[]` access, `()` call, etc. This is just advice, not a language rule, but it is good advice: conventional use of operators, such as `+` for addition, can significantly help us understand a program.*

Add following binary arithmetic operators to the given Health class and decide whether you should use member functions or non-member functions. If one needs to be non-member then it is good practice to keep the functions symmetric.

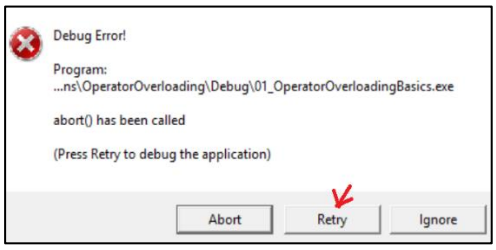
Arithmetic operators

These operations should be possible:

1. Health-object += integer
2. Health-object -= integer
3. Health-object = integer + Health-object
4. Health-object = integer - Health-object
5. Health-object = Health-object + integer
6. Health-object = Health-object - integer

Implement the functions in this same sequence, because the given tests are also written in this sequence. Work step-by-step, when you have implemented an operator, then test it by uncommenting the related test-code (in TestHealthClass function). When the test succeeds then **add your own code that uses** this operator (in UseHealthOperators function). Only when this is done, proceed with the next operator.

If one of the tests gives an unexpected result, then a **Debug Error window** appears. Pushing the Retry-button, leads you to the test that caused this message.




In the end you should have a console with all tests (6) reported as being succeeded.

```
==> 1. Test of: Health += int
ok

==> 2. Test of: Health -= int
ok

==> 3. Test of: int + Health
ok

==> 4. Test of: int - Health
ok

==> 5. Test of: Health + int
ok

==> 6. Test of: Health - int
ok
```

**When you have finished this exercise, and passed all the test cases, still ask the lecturer for feedback to verify your Health class code, not everything can be verified by tests.**

### 3.1.4. Container class with subscript [] operator

The subscript operator

#### a. Add the Container class

In this assignment you'll add subscript[] operator overloading to the Container class of previous lab. Add your Container class files to this project.

Those who didn't succeed in defining the Container class in that lab, can use the given Container class. Have a look at it, you should understand the code.

**Building and running your project shouldn't give any problems.**

#### b. Operator[] to get an element from the container

Add this operator to the Container class and then uncomment the related test code in the given **TestContainerClass** function, build and verify that no errors occur when the test is executed.

Then add your own code that uses this operator. Only when this is done, proceed with the next step.

#### c. Operator[] to change an element in the container

Adapt the operator[] in the Container class so that it can be used to **change** an element in the Container. Then uncomment the related test code in the given

**TestContainerClass** function, build and verify that no errors occur when the test is executed.

Then add your own code that uses this operator. Only when this is done, proceed with the next step.

### d. Operator[] to get an element from a const container

Add another operator[] function to the Container class so that it can be used to get an element from a const Container. Then uncomment the related test code in the given **TestContainerClass** and **PrintContainer** functions, build and verify that no errors occur when the test is executed.

In the end you should have 3 succeeded tests.

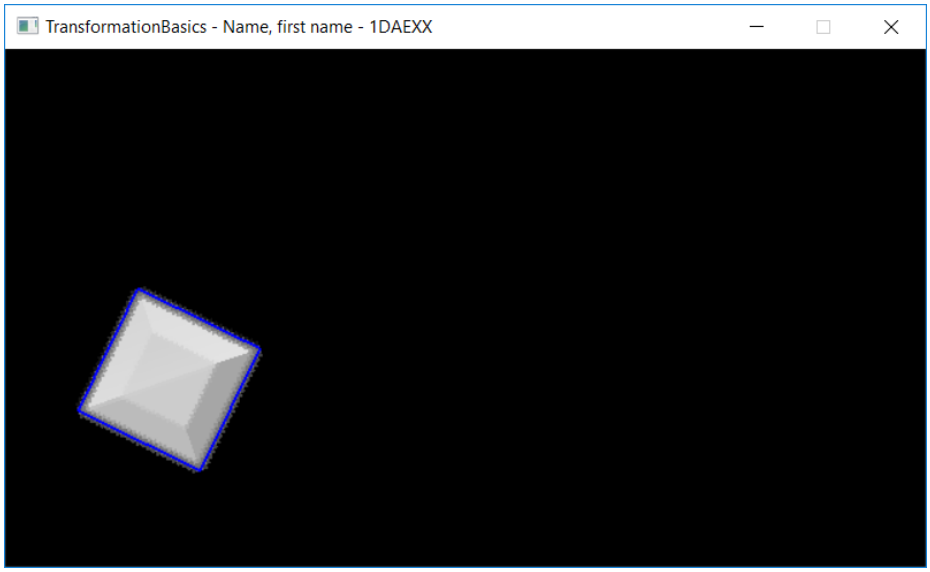
```
==> 1. Test of: Container[idx] to get an element
ok

==> 2. Test of: Container[idx] to assign a new value to an element
ok

==> 3. Test of: Container[idx] to get an element of a const Container
5 19 19 3 5
ok
```

**When you have finished this exercise and passed all the tests, ask the lecturer for feedback to verify your Container class code.**

## 3.2. TransformationBasics




### 3.2.1. Create the project

Add a new **framework project** with name **TransformationBasics** to the **W05** solution and set it as StartUp project.

Adapt the window title.

### 3.2.2. General

In this project you learn:

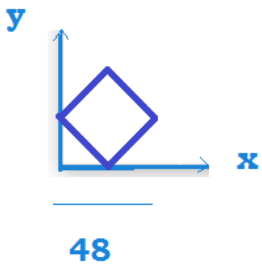
- How to use the given **Matrix2x3** functionality to transform vertices.

- How to indicate which **transformations** (translation, rotation or scaling) you want to apply to textures by changing the OpenGL **model matrix**.

More particularly you rotate, translate, and scale a diamond and detect whether it has been clicked.

### 3.2.3. Draw the diamond vertices

Define an array - m_Vertices - of 4 Point2f elements, and fill it with the values of the 4 vertices of a diamond: width and height are both 48 pixels and position it at the origin, like this:




Draw it in blue

Tip: use utils::DrawPolygon.

### 3.2.4. Transform the diamond vertices

#### a. Verify the transformation keys

By pressing keys, the user can transform the diamond. It can rotate, translate, or scale it. Define some data members that keep the actual transformation state. You need the angle, the scaling factor, and the translation vector. We'll use the same scaling factor in x and y directions.

Update these data members when the transformation keys are pressed.

- a- and d-keys change the angle (+= 60 * elapsedSec or -= 60 * elapsedSec). **The diamond should rotate around its center**.
- w- and s-keys change the scale factor (*= 1 + 3 * elapsedSec or /= 1 + 3 * elapsedSec)
- Arrow keys change the translation (+= 120 * elapsedSec or -=120 * elapsedSec)

```
a: Rotate ccw
d: Rotate cw
w: Zoom in
s: Zoom out
Arrows: Translate
```

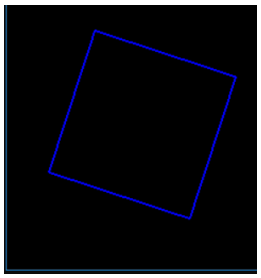
Recall that with the function **SDL_GetKeyboardState** one can get the array with the key states, e.g.:

```
const Uint8 *pKeysStates = SDL_GetKeyboardState( nullptr );
```

And the `SDL_SCANCODE_...` constants are the indexes in this array, e.g. next expression is true when the d-key is down:

```
pKeysStates[SDL_SCANCODE_D]
```

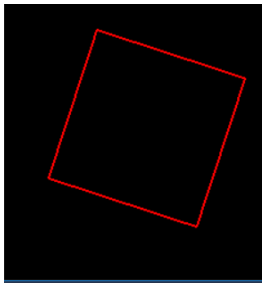
**b. Transform the vertices**



Define a second array - m_TransVertices - of Point2f elements to hold the transformed vertices. And each time a transformation change happens, calculate the transformed vertices starting from the original ones and using the Matrix2x3 functionality.

Draw the transformed vertices.

### 3.2.5. Select/deselect the diamond



When the LMB is clicked inside the transformed diamond then a selected – red - diamond is drawn, when it is clicked again the unselected – blue - diamond is drawn.

Tip: use utils::IsPointInPolygon

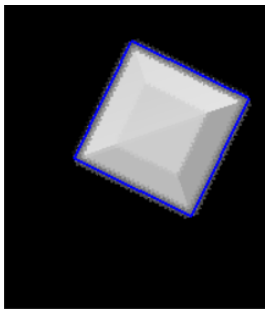
### 3.2.6. Draw and transform the diamond texture

Now let's see how we can draw a transformed texture.

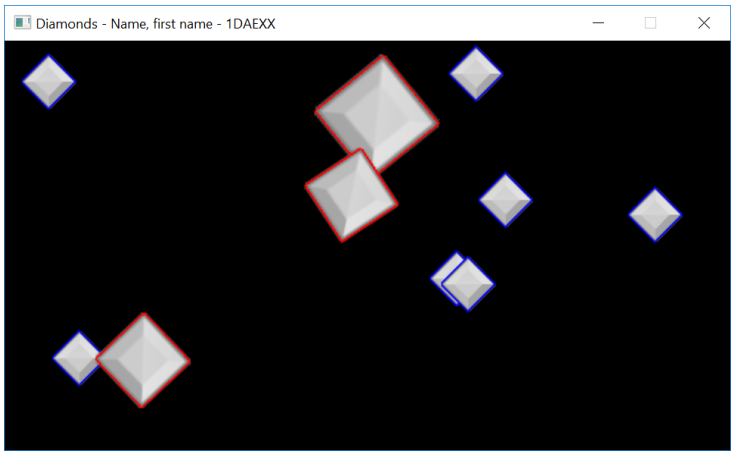
Create a Texture object using the given diamond image and draw it also in the origin of the window, before drawing the polygon, like this:

Transform the texture as well. After having drawn the Diamond texture at its starting position, draw it a second time after having changed the OpenGL model matrix using the angle, scale and position.

The picture below shows a translated, scaled and rotated diamond.



## 3.3. Diamonds




### 3.3.1. Create the project

Add a new **framework project** with name **Diamonds** to the **W05** solution and set it as StartUp project.

Delete the generated file **Diamonds.cpp**.

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

Copy and add the framework files.

Adapt the window title.

### 3.3.2. General

As an extra practice on defining classes and creating objects, let's define a Diamond class that implements the same transformation functionality as in one of the previous exercises.

### 3.3.3. Diamond class

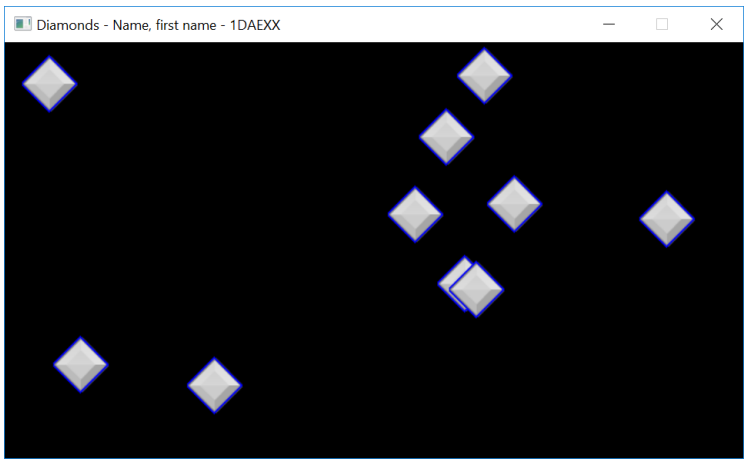
Add a new Diamond.h and .cpp file.

This class should provide following functionality:

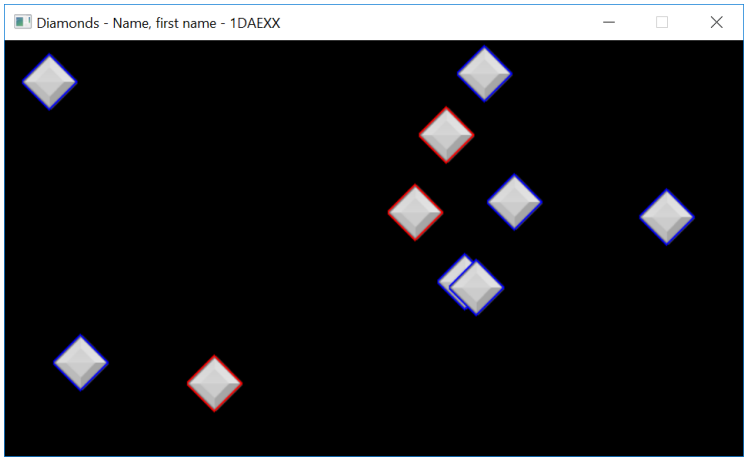
- When making an object the position can be specified.
- It can draw itself
- It is selectable
- Transformations can be added: translation, rotation, scaling. However only have effect when the diamond is selected.
- The transformations and selection state can be reset.

### 3.3.4. Create Diamond objects

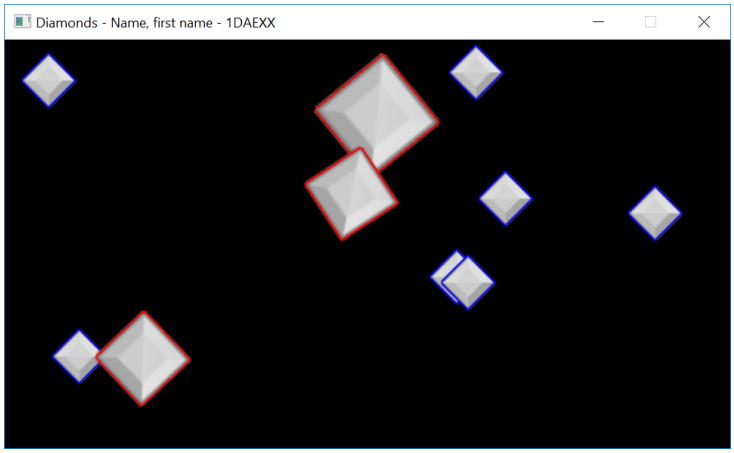
In the Game class make some Diamond objects positioning them randomly but inside the window boundaries and draw them. You can store the pointers in a std::vector.



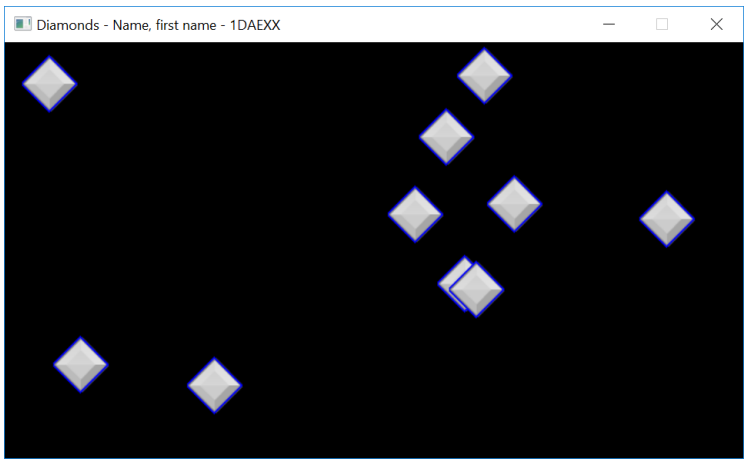

Select/deselect with the LMB. In the picture below 3 are selected.




Apply transformations using the same keys as in previous exercise.




When the r-key is pressed, then reset the transformations of the diamonds and deselect them.




## 3.4. MiniGame

Go to the Minigame assignment and implement part 3.5.

In this part you can:

- Add a **platform** to stand on and jump through.
- **End Game** sign, touching it ends the game.

Go to the Minigame assignment and implement part 3.6.

In this part you can add a **HUD**

Go to the Minigame assignment and implement part 3.7.

In this part you add **sound**.

# 4. Submission instructions

You have to upload the folder *1DAExx_05_name_firstname, however first clean up each project. Perform the steps below for each project in this folder:*

- In Solution Explorer: Select the solution, RMB, choose **Clean Solution**.
- Then **close** the project in Visual Studio.
- Delete the .vs folder.

Compress this *1DAExx_05_name_firstname* folder and upload it before the start of the first lab next week.

# 5. References

## 5.1. Operator overloading

### 5.1.1. General

http://en.cppreference.com/w/cpp/language/operators

http://www.learncpp.com/cpp-tutorial/91-introduction-to-operator-overloading/

### 5.1.2. Arithmetic operators

http://en.cppreference.com/w/cpp/language/operators paragraph **Binary arithmetic operators**

https://en.cppreference.com/w/cpp/language/operator_arithmetic

### 5.1.3. Assignment operators

https://en.cppreference.com/w/cpp/language/operator_assignment

### 5.1.4. Comparison operators

https://en.cppreference.com/w/cpp/language/operator_comparison

### 5.1.5. The subscript operator

http://en.cppreference.com/w/cpp/language/operators paragraph **Array subscript operator**

http://www.learncpp.com/cpp-tutorial/98-overloading-the-subscript-operator/

### 5.1.6. Stream insertion operator

http://en.cppreference.com/w/cpp/language/operators paragraph **Stream extraction and insertion**