CGGE2 – CG Game Engine 2

Based on version 1.0

1.0 Introduction

CGGE2 offers an easy way to create games in Java. It is simple and straight forward.
First, you must download one of the latest builds of the API and include it into your project. Based on the IDE or editor you are using, the approach may vary a little.

In this documentation, we will dive into every single aspect of the API so that you are going to be able to use all of its features.

1.1 Making your first game

```
public static void main(String[] args) {
    GameInstance game = new GameInstance();
}
```

Make sure, you have *de.cg.cgge.game.GameInstance* imported.
And that's it… Now you have successfully created your first game.
The *GameInstance* object serves as the core of every game. It contains a *Drawer*, which then contains a *Window* and a *Room*, in which the game objects are going to be placed in.
Each room has its own *Clock* and an own *ObjectManager* which together are responsible for the whole game's logic.

This might seem complicated at first, but you don't really need to understand what exactly is going on for now.

1.2 The config file

To fully setup your game, you would also need a configuration file. For that simply create a *"rsc//config.data"* file and add it as a parameter to your *GameInstance*.

```
public static void main(String[] args) {
    GameInstance game = new GameInstance("rsc//config.data");
}
```

In the config file, you can specify various arguments for your game:

```
framerate: 60
width: 1920
height: 1080
title: My Test Game
taskbar: false
```

These arguments can only be set once the game launches.
Note that width and height only define the resolution of the windowed mode, not full screen mode.

CGGE2 – CG Game Engine 2

Based on version 1.0

2.0 Your first GameObject

Now that you have your game set up, we can proceed with creating your first *GameObject* instance.

```
import de.cg.cgge.game.GameObject;

public class TestObject extends GameObject() {

    //Now you should implement the constructor
}
```

Now you can override the draw method and draw your first game object to the screen.

```
@Override
public void draw(Graphics g) {
    g.setColor(Color.WHITE);
    g.fillRect(50,50,50,50)
}
```

It will now display a rectangle at the specified position. The draw method is called sixty times a second. Before all the draw methods are called the screen gets cleared entirely. If you want to draw after every other object's draw method has been called, then you should use the *postDraw(Graphics g)* method. Doing so is recommended when dealing with GUI.

Another trait of game object is that they have *float x, y* and *int w, h* already pre-defined.
It's recommended to use these to draw your object as the position should update when the position of the object updates as well as these values are tied to the physics engine.
So let's update the code:

```
public TestObject(Room room) {
    super(room);

    this.x = 50;
    this.y = 50;
    this.w = 50;
    this.h = 50;
}

@Override
public void draw(Graphics g) {
    g.setColor(Color.WHITE);
    g.fillRect((int) x,(int) y, w, h);
}
```

To create your object you simply go back to your main method and write

```
TestObject testObject = new TestObject(game.getRoom());
```

The *getRoom()* method returns the currently active room, in which the object should be placed in.

The following table lists all the methods that can be overridden:

protected void create()	Is called, when the object is added to the main loop
public void step()	Is called every tick. (That is every frame)
public void preStep()	Is called every tick, but before any step() methods are called
public void draw(Graphics)	Is called every tick. Runs on the draw thread and its purpose is to make your objects visible on the screen. Here is more information on *Graphics*.
public void postDraw(Graphics)	Is like draw(Graphics), but is called AFTER all the draw() methods. Is recommended for GUI.
public void mouseClicked(MouseEvent)	Is called when the mouse is clicked
public void mousePressed(MouseEvent)	Is called, once the mouse is pressed
public void mouseReleased(MouseEvent)	Is called, once the mouse is released, after it was pressed
public void keyTyped(KeyEvent)	Is pressed, when letters are typed. e.g. When the user presses SHIFT + S, it is registered as an uppercase 'S'

All the other methods should not be overridden.

There are also the booleans *solid,* which should be set to false, if you don't want your object to collide with other objects, and there is *visible,* which makes the main loop skip the draw() and postDraw() methods if set to false.

2.1 Making your object move

In order to let your object move around based on user input, you have to use the *KeyManager* class. It contains a boolean array, that stores, when each key is pressed.
You now have to ask for those booleans and fetch whether the keys are pressed. When they are pressed, you have to raise or lower the corresponding x or y values accordingly.

```
KeyHandler key = new KeyHandler(room.getGameInstance().getDrawer());

@Override
public void step() {
    if (key.keyPressed(KeyEvent.VK_W)) {
        y-=5;
    }
    if (key.keyPressed(KeyEvent.VK_S)) {
        y+=5;
    }
    if (key.keyPressed(KeyEvent.VK_A)) {
        x-=5;
    }
    if (key.keyPressed(KeyEvent.VK_D)) {
        x+=5;
    }
}
```

Note, that these methods return true as long as the button is pressed. So when you try to toggle something with a key, you are going to toggle it repeatedly. This can be solved with the following structure.

```
boolean lock = false;

if (key.keyPressed(KeyEvent.VK_SPACE)) {
    if (!lock) {
        //DO YOUR ACTION HERE
        lock = true;
    }
} else {
    lock = false;
}
```

2.2 Movement with Physics

Instead of adjusting the x and y positions manually you should rather use the integrated physics engine. Currently there are two physics classes which are *Gravity* and *Mover*. We will focus on the latter one for now.

The Mover, as the name might already imply, moves your object around for you. It does that while also taking collisions with other objects into account that have *solid* set to *true*. In order to gather collision information, it uses a *Collider* instance which then can check for box collisions with other objects.

Every GameObject has an ArrayList of Physics (with a default size of zero, so there is little memory penalty). To add a Mover, you must create one and add it to that list.

```
private Mover mover;

public TestObject(Room room) {

        ...

    mover = new Mover(this); //Creates a mover instance

    addPhysics(mover);
}
```

You can now set the x and y speeds of the Mover to move the object. You can also optionally set accelerations for both directions to implement force (> 1) or simulate ice (< 1). To let the movement halt after the button is released, you should leave it at 0.
To update the physics, you need to call *updatePhysics()* after every step.

```
@Override
public void step() {
    if (keyUp) {
        mover.setYspeed(-5); //Going up the screen
    }

    if (keyDown) {
        mover.setYspeed(5); //Going down the screen
    }
```

```
    if (keyLeft) {
        mover.setXspeed(-5); //Going to the left
    }

    if (keyRight) {
        mover.setXspeed(5); //Going to the right
    }

    updatePhysics();
}
```

2.3 Making movement framerate independent

So right now, your physics update at a given framerate. But what if the Game is not able to sustain the target framerate? Well then your game is going to be updated less frequently, so your game is going to get slower. If your target is let's say 60FPS but the user only reaches 30FPS, he will play at half the intended speed.

To fix that, the engine offers an *adjusted delta time.* Delta time is the time in seconds, the frame took to be drawn. It is adjusted to be relative to the framerate. So when your target is 60FPS and the frame takes just the right amount of time, then the delta value is going to be 1.0. But if the framerate drops to 30FPS, you would get a delta value of 2.0. Multiply that with your speed values and your game is *Framerate Independent.*

```
float delta = Physics.adjustedDeltaTime(room.getGameInstance());
float speed = 5f*delta;
```

But of course, there is one catch. And that is collisions. As the speed is doubled, it moves across more quickly. And as the framerate is halved, it checks for collisions less frequently. That means, that if a point moves at the speed of 40*deltaTime towards a wall 50 pixels thick, it will most likely miss the collision, as it's x position is raised to 80 (at a halved framerate) which then would be behind the wall. So, it's only recommended with smaller speeds, but should definitely be implemented.

Also, there is a small performance penalty, as there is one more floating point operation per speed value per frame. That can stack up pretty quickly.

2.4 Using sprites

In order to use sprites with your game objects, you must create them first.

```
//public Sprite(String path, int width, int height, int rotation);
private Sprite sprite = new Sprite("rsc//sprite.png", 64, 64, 0);
```

Remember to put your image file into the right directory. If you want to clone an already existing sprite you can just do that as follows:

```
//public Sprite(Sprite existingSprite);
private Sprite sprite = new Sprite(existingSprite);
```

The sprite of course has all sorts of getters and setters for all of its values. We won't be diving into each of them. For more information, just read the javadocs.

If you now want to draw the sprite to the screen, head into your *draw(Graphics g)* method and do as follows:

```
sprite.draw((int) x, (int) y, g);
```

If you set a rotation, then you should also specify a center. This can be done with the sprite's *setCenter(int x, int y)* method.

2.5 Collider

If you wanted to check for collisions yourself, then it's recommended to use a *Collider* instance as it offers quite a lot of features.

It differs between 2 major kinds of collisions, the box and circle collisions, and between 3 minor kinds of collisions each. These are *solid, unsolid* and *general*. But as of now, the circle collisions are not yet available, but going to come in a later patch. And with *getLastCollision()* you can get the game object the object the collider belongs to last collided with.

An example:

```
private Collider collider = new Collider(room, this);
int score = 0;

@Override
public void step() {

        ...

    //Check if there is a collision with an object that is not solid
    if (collider.checkUnsolidBoxCillision(x,y,w,h)) {
        GameObject lastCollision = collider.getLastCollision();

        //Check if the object, the GameObject collided with is of instance coin
        if (lastCollision instanceof Coin) {
            score++;
            lastCollision.destroy(); //Destroys the coin on pickup
        }
    }
}
```

Keep in mind that when an object collision is detected, the collider remembers the index of the object in the object list. When you later call *getLastCollision()* it gathers the same index from the object list. This is the fastest approach, but it has one disadvantage. When it takes a little to call the method, the object might already be killed which is going to result in either unexpected results or an error.

If you wanted to get access to the object list yourself, you would have to access the *ObjectManager* of the room. It's easily accessed like this:

```
//ObjectManager
ObjectManager om = room.getObjectManager();
//ArrayList of objects
ArrayList<GameObject> objects = om.getObjects();
```

But don't ever modify it directly. Editing it is possible through the *addObject(GameObject)* and *removeObject(GameObject)* methods from the object manager. It won't edit the list immediately but instead schedule it to a point in time where it won't cause any concurrent modification issues.

But also, this is not really necessary as the constructor of the game object and it's destroy method do the job for you.

2.6 Gravity

There is one more kind of Physics class that is related to the mover. In fact, it relies on a mover being implemented as well as it manipulates it directly to let the object fall 'down'.

Let's just take a look at the code of the de.cg.cgge.physics.Gravity class itself to look at how to make physics objects yourself.
The two most important methods are its constructor and the *update* method that it inherited from the *Physics* class.

```
//Extending Physics
public class Gravity extends Physics {

    /*
        Defining all the variables needed
    */
    private float force = 1.0f; //The force that represents gravity
    private final float beginForce; //Storing the force to reset it
    private float acceleration = 1.0f; //To accelerate gravity


    private Mover mover;
    private Collider collider;

    //Constructor sets up all the variables
    public Gravity(GameObject obj, float force, Mover mover) {
        super(obj);     //Calling the Physic's constructor

        this.force = force;
        this.beginForce = force;

        this.mover = mover;
        mover.setYacceleration(1.0f);

        //Creating a collider
        this.collider = new Collider(obj.getRoom(), obj);
    }

    @Override
    public void update() {
        //Letting the object fall down
        mover.setYspeed(mover.getYspeed()+force);

        //Accelerating gravity
        force*=acceleration;

        //Resetting gravity once it hits the ground
        if (collider.checkSolidBoxCollision(obj.getX(), obj.getY()+1, obj.getWidth(),
            obj.getHeight() || mover.getYspeed() > 0) {

            force = beginForce;
        }
    }

    . . .

}
```

As you can see it's not that difficult, but it still takes a little bit of work away from you.
It's basically just the object moving with a velocity of *force,* accelerated by *acceleration*. Once it hits the ground, it resets the *force* value.

I guess you now also know how to add it to your game object. Just create an instance of it and call *addPhysics(gravity).* But don't forget to create a *mover* instance and to add it to the list as well.

2.7 Creating sounds

Creating sounds is easily achieved by defining *Sound* objects.

```
//Already loads the sound into memory
Sound sound = new Sound("rsc//sound.wav");

sound.play(); //Starts the sound
```

Make sure to enter the right path to the sound as an argument. To stop the sound, simply call the *stop()* method. If you wanted to loop the sound, there is the *loop(int times)* method. If you wanted an infinite loop, you'd set *-1* as a parameter.

To reset the sound, you call *reset()* and to delete it from memory you say *erase()*. You can also set and get the current position of the sound (in milliseconds) to manage it more precisely.

2.8 Animations

Animations may sound scary at first, but they are actually quite simple. Instead of a sprite, you just use an *AnimatedSprite*. To animate your sprite you first have to create it.

```
//The first three args are just like the sprite
//The following ones are all the image paths
AnimatedSprite as = new AnimatedSprite(50, 50, 0, "rsc//img1.png",
"rsc//img2.png", "rsc//img3.png");
as.load();
```

All the animated sprite does is it switches in a certain range between all the images that it contains. Very important: Unlike the sprite, animated sprites must be loaded into memory first.

To start the animation you just type

```
int startImage = 0;
int lastImage = 2;
int timeInterval = 100; //The time in milliseconds between each image
as.startAnimation(startImage, lastImage, timeInterval);
```

The rest is the same as with sprites. You can draw it via the same *draw(int, int, Graphics)* method.

3.0 Camera

Another feature of the engine is the camera. It allows you to move your objects in space while keeping them in view. The basic camera implementation is quite simple. It has a target object, a position and two paddings, one for x and y. Once the targets position crosses the padding, the camera moves at a certain speed to that direction.

With a *CameraRenderer* all the draw calls are now adjusted to the camera's position. But let's setup the camera first.

```
room.getCamera.setObjectToFollow(this);
room.getCamera.setXpadding(200);
room.getCamera.setYpaddding(100);
room.getCamera.setSpeed(12);
```

We are just accessing the default camera of the room. Alternatively, we could also create a completely new camera instance. You can also zoom into the camera with *setZoom(float)*.

3.1 Camera Renderer

To do draw calls adjusted to the camera you must create a *CameraRenderer* inside your *draw(Graphics)* method.

```
@Override
public void draw(Graphics g) {
    CameraRenderer cr = new CameraRenderer(g, room.getCamera());
    cr.fillRect((int) x, (int) y, w, h);
}
```

The camera renderer contains pretty much all the necessary methods to properly draw to the screen. If still necessary, you can access the adjusted values through the *getAdjustedX(int)* and *getAdjustedY(int)* methods.

3.2 Switch to full screen

In order to switch into full screen mode, you have to access the *Window* of the *Drawer* of the *GameInstance*.

```
game.getWindow().switchFullScreen();
```

This cannot be done, when the taskbar is activated, as it would throw an *IllegalStateException.*

3.3 Switch game rooms

```
Room room = new Room(game);

//Init objects for the room here

game.getDrawer().changeRoomSafely(room);
```

4.0 Easily reading files

Though you can still just use a file library of your choice or do it the standard way, the engine offers a simple solution. *GameFile*s.

```
try {
    GameFile gf = new GameFile();
    gf.load(); //Load it into memory
} catch (IOException e) {
    System.out.println("Failed to load file.");
}
```

The file contains a *FileContents* instance, which contains all the lines of the file in an *ArrayList*.
It is returned by `gf.getContents();` and contains all sorts of useful methods:

`String[] get()` `ArrayList<String> getArrayList()`	Returns an Array/ArrayList of lines
`void set(String[])` `void set(ArrayList<String>)`	Set the lines of the contents
`void print()`	Prints every line
`void edit(int, String)`	Edit a specific line
`void append(String)`	Append a line
`void setTableSplitter(String)`	Sets the symbol that must be between table columns to separate them
`String getFromTableSection(int, int)`	Returns the String that is at exactly this position
`String[] getFromTableRow(int)`	Returns the entire row as an Array

`String getFromKeyword(String)`	Returns the value that is behind this keyword. The syntax is as follows and may NOT differ:
	
	`a keyword: this is the value`
	
	e.g:
	`getFromKeyword("a keyword");`
	returns "this is the value"

To save a file you can call its *save()* or *save(File)* methods.

4.1 TileMap

Tile maps are easy ways to create huge maps. It has an image, a so-called tile set, which is separated into many sections. The tile map now specifies where each section should be drawn at.

To make it all easier, you should use the TileEditor, that belongs to the engine. (*Currently experimentally available on GitHub*)

A tile map is stored in a plain text file. The recommended extension is **.tilemap*.

The first line contains basic information about the tile map.

```
The Path to the tile set          The sizes of each tile   width      height
"rsc//tileset.png"   ,    32   ,    2    ,    2
```

The second line contains the first layer to be drawn. It is drawn before the GameObjects.
Each tile of the tile set gets a number. These are then drawn at the respective x and y positions.
If you wanted an empty space, you would just type -1

```
x1, y1     x2, y1        x1, y2        x2, y2
4    ,    7    ,    9    ,    10
```

And the third line contains the second layer. It follows the same scheme, but it is drawn after all the GameObjects.

```
-1   ,    -1   ,    6    ,    -1
```

The fourth line contains the collision layer. There an *x* stands for collision and an *o* stands for no collision. The collisions can later be gathered by calling *isCollision(int, int).*

```
x    ,    o    ,    o    ,    x
```

And the last line is the action layer. It allows the creator of the tile map to add additional information to each tile.

```
spawn:test   ,    x    ,    x    ,    walk:teleportOrSomething
```

The final file should now look like this:

```
"rsc//tileset.png",32,2,2
4,7,9,10
-1,-1,6,-1
x,o,o,x
spawn:test,x,x,walk:teleportOrSomething
```

To use this tile map, you have to do the following:

```
String path = "rsc//tilemap.data";
int scale = 2; //Scale is the factor by which the images are sized up
game.getRoom().initTileMap(path, scale);
```

And now the tile map should already draw to the screen unless you have messed something up.

4.2 Other useful methods

To get the framerate:

```
game.getCurrentFramerate();
```

The *MouseHelper* class contains mouse information necessary to create buttons:

```
MouseHelper mh = new MouseHelper(game);
int mx = mh.getMouseX();
int my = mh.getMouseY();
```

Join the official discord for more information: https://discord.gg/Nc9hPSH

Created on 04.02.2020 by CG John