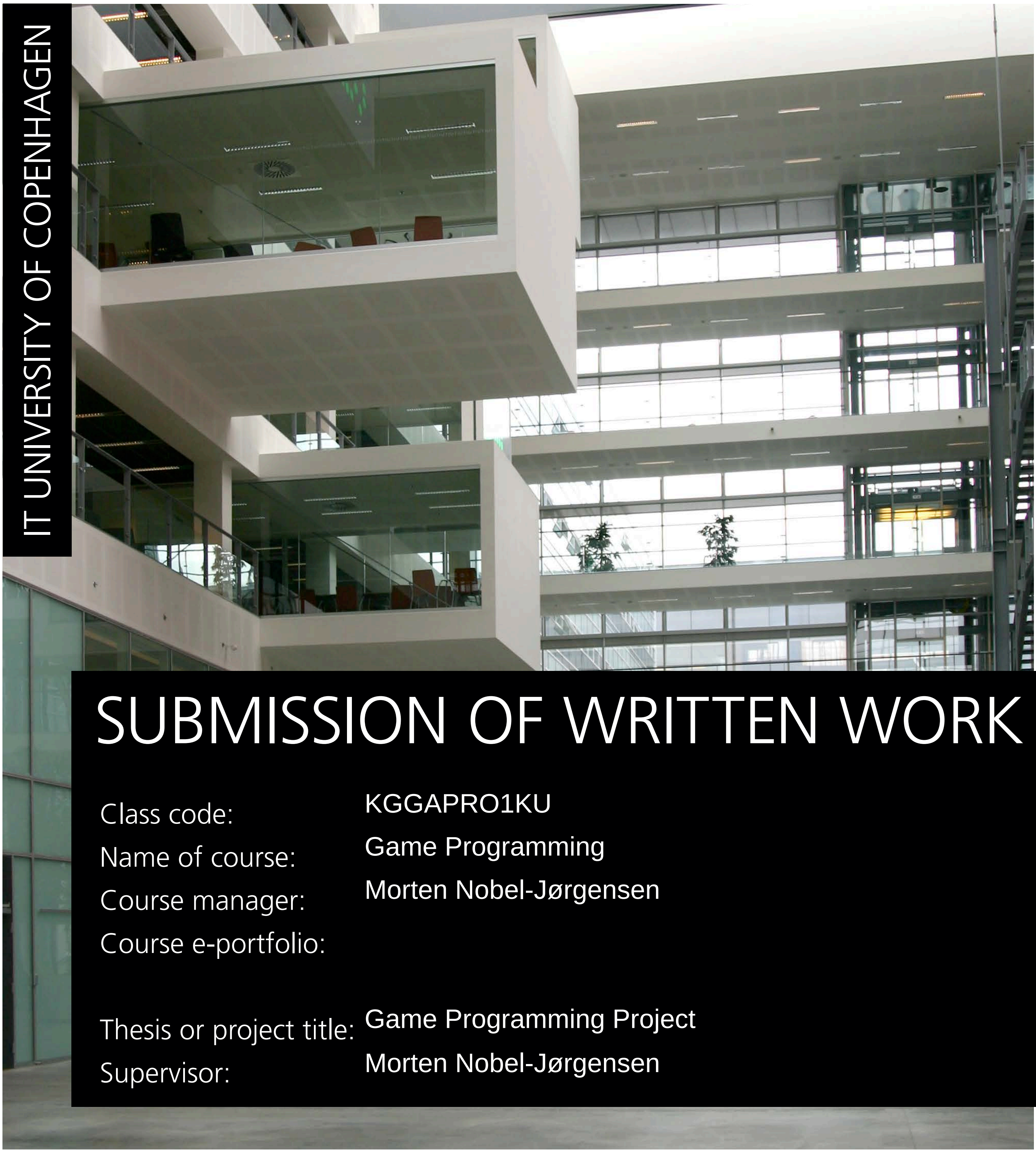


SUBMISSION OF WRITTEN WORK

Class code:	KGGAPRO1KU
Name of course:	Game Programming
Course manager:	Morten Nobel-Jørgensen
Course e-portfolio:	
Thesis or project title:	Game Programming Project
Supervisor:	Morten Nobel-Jørgensen

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Simon Bøg	22/10-1987	sibc@itu.dk
2.	Joseph McGeoch Osborne	18/05-1993	jomc@itu.dk
3.	Victor Olsson	04/08-1993	viol@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Introduction

Our project is a 2D sprite based platformer game engine using the Simple Render Engine (SRE), with a dynamic event based update system and a physics system, with included collision detection, based on Box2D.

The game engine features the following capabilities:

- A tile based level system based on external json files.
- A Sprite Atlas system based on external json and png files.
- Any correctly formatted level or sprite atlas files can be loaded, through the applications command line interface.
- Tiles is based on a GameObject -> Component system. Where one tile/GameObject can have multiple components attached. Components include, but are not limited to: CharacterController, PhysicsBody, SpriteRenderer and Transform.
- PhysicsBody component capable of being both static and dynamic with desired density. Collision detection between both types is possible.
- Dynamic event system for both keyboard input, update, physics and rendering. Components automatically subscribe or unsubscribe to their needed events, based on their Game Objects active status.

The games that users can create using our system is constrained to 2D platforming games. The example game that we have made is a 2D physics based platformer. You, the player, is a space traveling alien trapped on a hostile world. You have to search the world for the energy crystals that power your ship so you can return to your native planet. The player navigates the level by timing their jumps so they do not get hit by the falling obstacles. They also have to use their ingenuity to solve puzzles to complete certain levels.

Architecture

The architecture of the engine is Component-Entity based and event driven. A Component-Entity framework was selected in order to improve separation of concerns. Event driven systems were used in order to reduce overall coupling.

Overview

UML class diagrams have been prepared for all aspects of the engine built on top of SRE and its dependencies. Some type from these libraries appear in these diagrams, without the full scope of their identifiers.

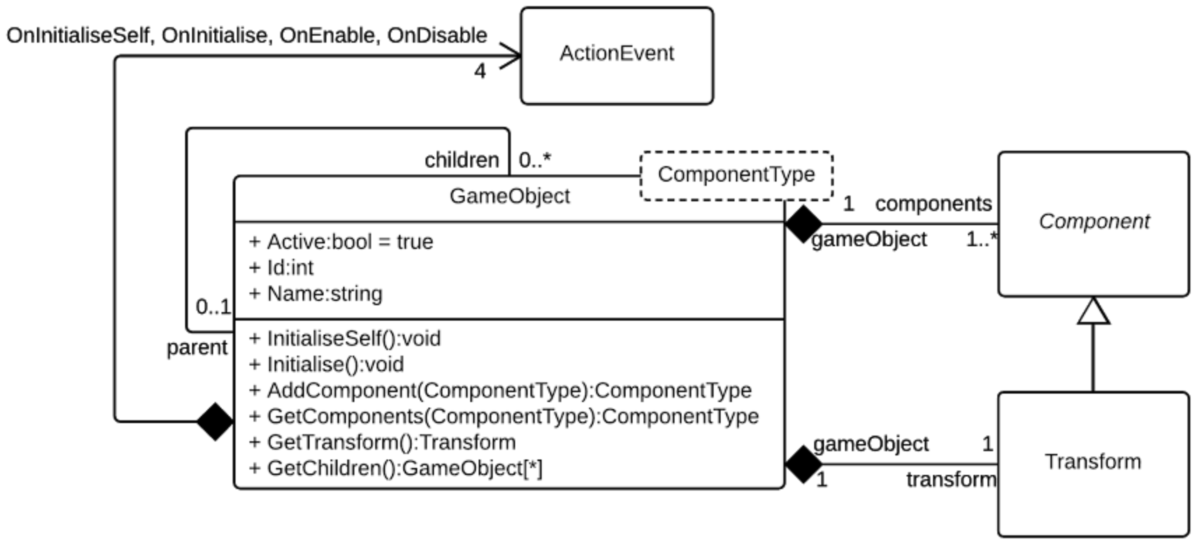


Figure 1: *Class diagram showing GameObject relationships to its Components*

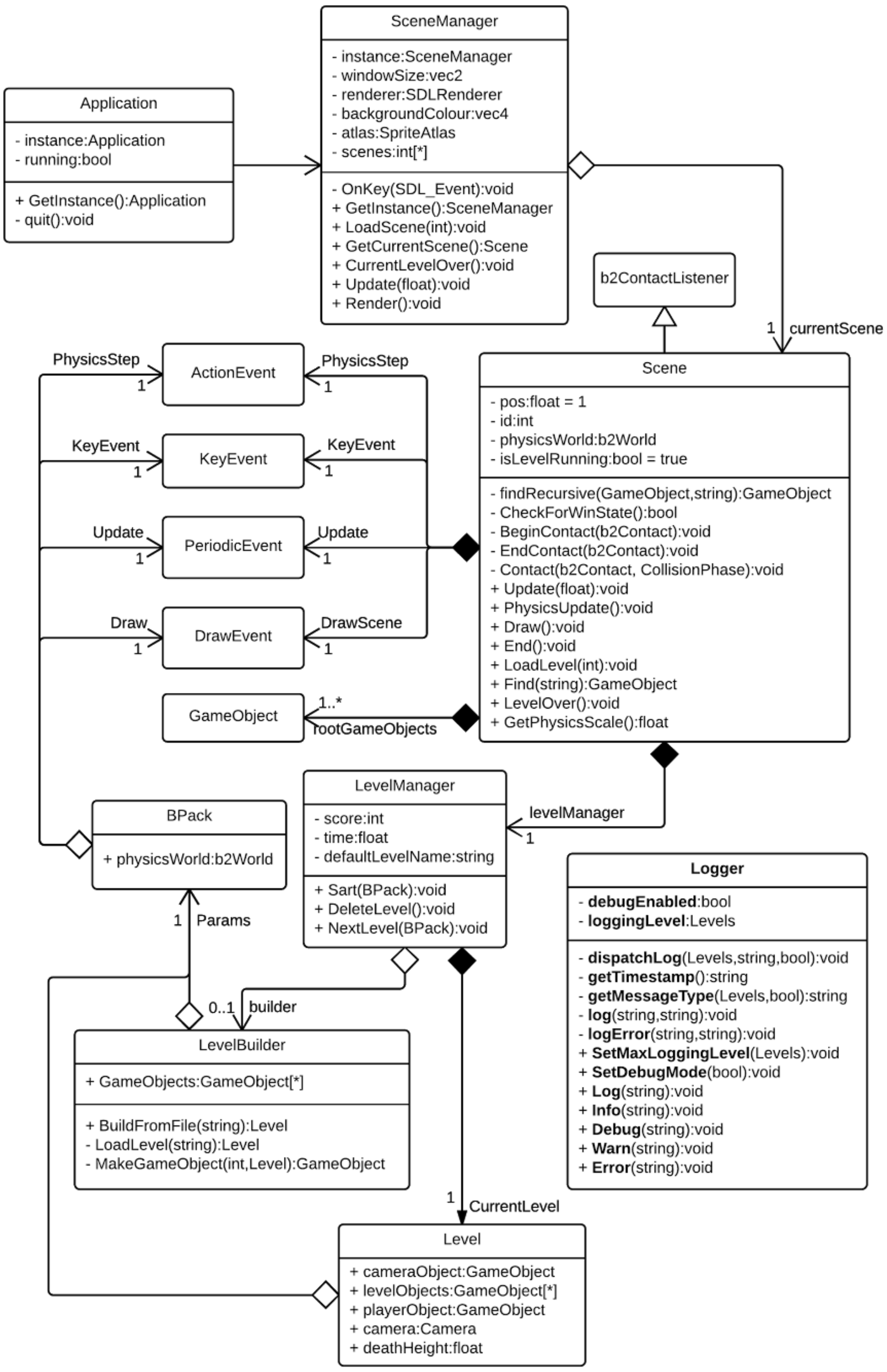


Figure 2: *Class diagram showing overall application composition*

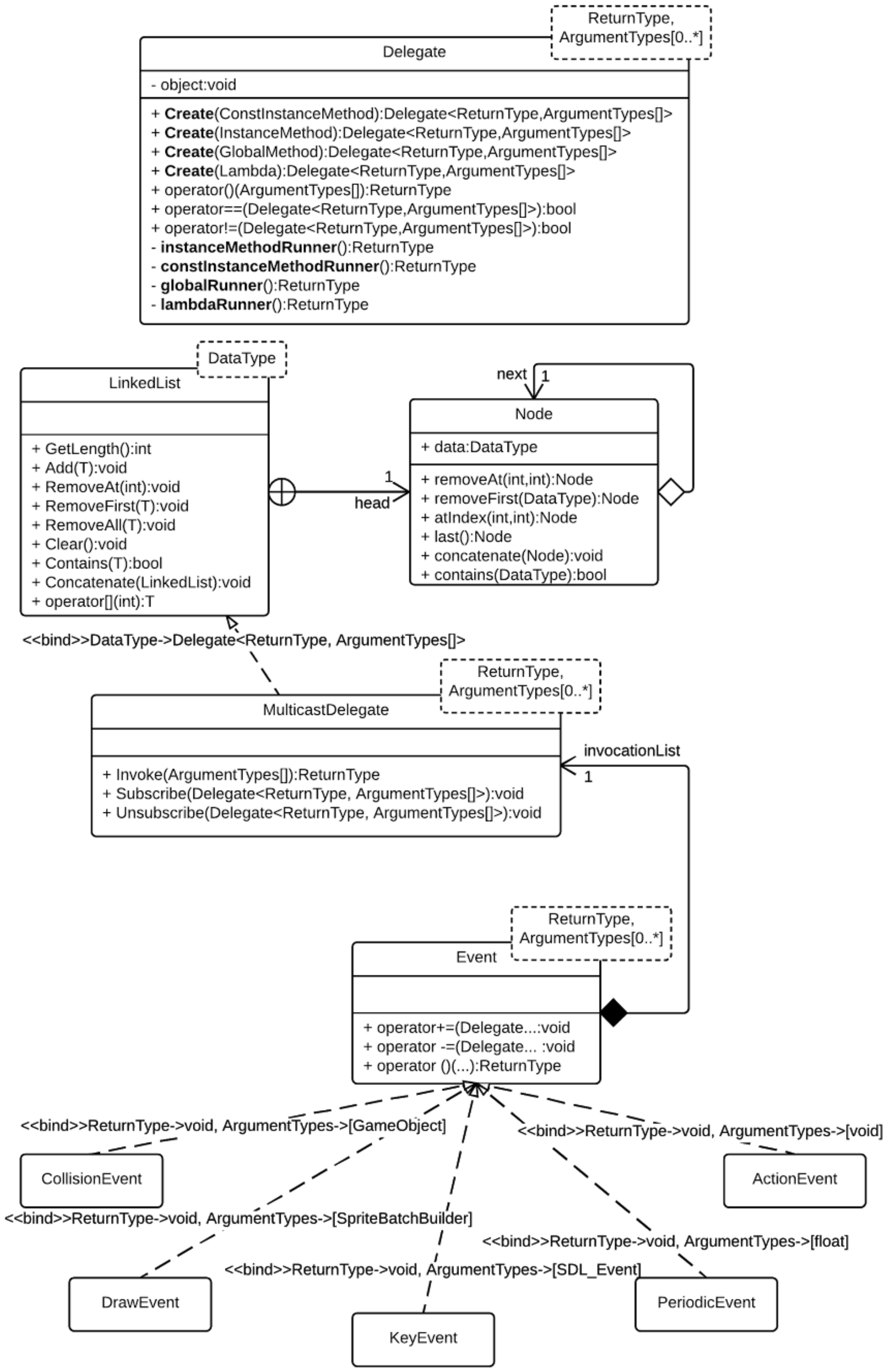

ReturnType, ArgumentTypes[0..*]
Delegate
- object:void
+ Create(ConstInstanceMethod):Delegate<ReturnType,ArgumentTypes[]>
+ Create(InstanceMethod):Delegate<ReturnType,ArgumentTypes[]>
+ Create(GlobalMethod):Delegate<ReturnType,ArgumentTypes[]>
+ Create(Lambda):Delegate<ReturnType,ArgumentTypes[]>
+ operator()(ArgumentTypes[]):ReturnType
+ operator==(Delegate<ReturnType,ArgumentTypes[]>):bool
+ operator!=(Delegate<ReturnType,ArgumentTypes[]>):bool
- instanceMethodRunner():ReturnType
- constInstanceMethodRunner():ReturnType
- globalRunner():ReturnType
- lambdaRunner():ReturnType
DataType
LinkedList
+ GetLength():int
+ Add(T):void
+ RemoveAt(int):void
+ RemoveFirst(T):void
+ RemoveAll(T):void
+ Clear():void
+ Contains(T):bool
+ Concatenate(LinkedList):void
+ operator[](int):T
1
head
next
1
Node
+ data:DataType
+ removeAt(int,int):Node
+ removeFirst(DataType):Node
+ atIndex(int,int):Node
+ last():Node
+ concatenate(Node):void
+ contains(DataType):bool
<<bind>>DataType->Delegate<ReturnType, ArgumentTypes[]>
ReturnType, ArgumentTypes[0..*]
MulticastDelegate
+ Invoke(ArgumentTypes[]):ReturnType
+ Subscribe(Delegate<ReturnType, ArgumentTypes[]>):void
+ Unsubscribe(Delegate<ReturnType, ArgumentTypes[]>):void
invocationList
1
ReturnType, ArgumentTypes[0..*]
Event
+ operator+=(Delegate...:void
+ operator -=(Delegate... :void
+ operator ()(...):ReturnType
<<bind>>ReturnType->void, ArgumentTypes->[GameObject]
CollisionEvent
<<bind>>ReturnType->void, ArgumentTypes->[void]
ActionEvent
<<bind>>ReturnType->void, ArgumentTypes->[SpriteBatchBuilder]
DrawEvent
<<bind>>ReturnType->void, ArgumentTypes->[float]
PeriodicEvent
<<bind>>ReturnType->void, ArgumentTypes->[SDL_Event]
KeyEvent

Figure 3: *Class diagram showing the Events used*

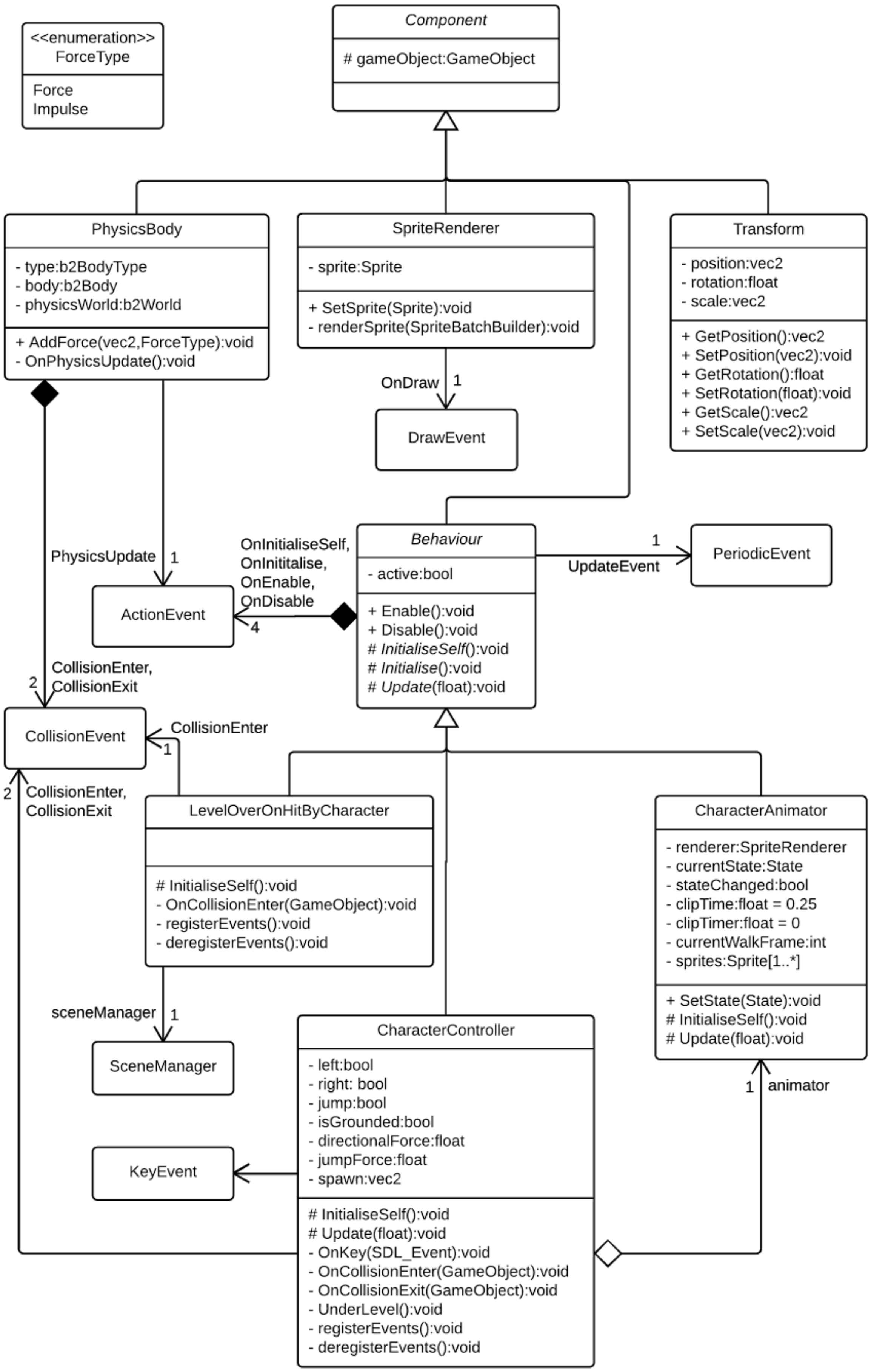


Figure 4: *Class diagram of Component and its child classes*

Implementation

In order to show each of the constructed classes, each of them is described below. In addition to this, each class is accompanied with a description of its interface.

Logger

A static class used to provide logging functionality to the engine. Creates a useful output stream of messages at various levels of severity to either std::clog or std::err as appropriate.

- **Methods**:
    - **SetMaxLoggingLevel**: Sets the maximum severity of received logs to output.
    - **SetDebugMode**: Sets debug mode, whether or not debug messages get printed.
    - **Log**: Lowest severity message, only printed when maximum logging level is all.
    - **Info**: Low severity message.
    - **Debug**: Debugging message. Only printed when debug mode is enabled. Same severity as Info.
    - **Warn**: A warning message.
    - **Error**: An error message. This may be either recoverable or fatal errors.

Application

Application is responsible for taking input from the player and starting the rest of the system based on this input.

- **Application** takes the input from the user regarding which world and sprite atlas the user wants to run and checks if these files exist. If they do not, the application will ask the user to reenter the filename. Once they are correct the application will create the SceneManager and give its constructor the desired world and sprite atlas. Writing “1” when asked will just run the default game with the default sprite atlas.

SceneManager

SceneManager is a static class which sets up the SRE and controls the current Scene. It also have difference variables that are reachable application wide.

- **SceneManager** is responsible for creating and controlling the current scene. It is also responsible for starting SRE and controlling its update loop. It sets SRE keyevent to onKey, frameUpdate to update and binds frameRender to render, on the current instance of SceneManager. The constructor ends by calling the LoadScene method with the users world and sprite atlas as arguments before starting the SRE eventLoop.
- **GetInstance** will return the current instance of SceneManager.
- **LoadScene** creates a sprite atlas from the spriteAtlas argument and calls LoadLevel on its currentScene with the user's world as argument.
- **onKey** checks if an incoming SDL_Event is the Escape key, if it is it will exit the application, if it is not it will send the event as an argument to the currentScene's KeyEvent method.
- **update** will call the method update on currentScene with the time since last update as an argument.
- **render** will call the method render on currentScene.
- **CurrentLevelOver** method will call LevelOver on the currentScene.

Scene

Scene is responsible for for starting the building of a new level, deleting the old level and running the event system.

- **Draw** creates a SRE renderPass with the camera from the levelManager's currentLevel as well as a SpriteBatchBuilder and calls the event DrawScene with the SpriteBatchBuilder as an argument. The result of this event call is then build into a SpriteBach and drawn with the renderPass. If physics debugging info is desired it will draw the physics colliders as lines with the renderPass.
- **update** calls the PhysicsUpdate method and calls the update event with time since last frame as the argument. If CheckForWinState method returns true, it will delete the old level and load the new level through the levelManager and the call InitialiseSelf and Initialise on every GameObject in the the current level
- **PhysicsUpdate** calls the current scenes physics worlds step function and then call the PhysicsStep event.
- **LoadLevel** sets Update, DrawScene, PhysicsStep and KeyEvent to the relevant event types, creates a new physics world and set itself as the ContactListener, and

levelManager’s Start method is called to set up the new level. Lastly it call InitialiseSelf and Initialise on every GameObject in the the current level.

- **Find** goes through every GameObject in the levelManager’s levelObject vector and will return the first GameObject with the name given as argument. If non exist in the vector, it will go through all of the GameObjects children
- **BeginContact** calls the Contact method with a reference to the b2Contact and Enter as the Collision Phase
- **EndContact** calls the Contact method with a reference to the b2Contact and Exit as the Collision Phase
- **Contact** calls the Collision method on the PhysicsBody component owning each of the two colliding fixtures with the other GameObject and the collision phase as arguments
- **findRecursive** returns a GameObject with the name passed as argument in the children of the GameObject passed as argument.
- **CheckForWinState** returns the variable isLevelRunning.
- **LevelOver** sets the variable isLevelRunning to false.

LevelManager

LevelManager is responsible for getting the current level build as well as keeping track of the of relevant information regarding this build level.

- **Start** this is used to initially setup the level, it sets initial parameters and calls the nextlevel method
- **DeleteLevel** this is used to delete a level, by looping through the level objects and deleting them.
- **NextLevel** is a method used to create the level builder, which returns the Level that is then set as the current level.

LevelBuilder

LevelBuilder is a factory and is responsible for building the desired Level and all of the GameObjects and their Components.

- **Level (struct)**
    - **levelObj** is a shared pointer to the actual level
    - **cameraObject** is a shared pointer to the GameObject the CameraController component is attached to.

    - **levelObjects** is every gameObject present in the level
    - **deathLevelInY** is the levels death zone value. Every PhysicsBody component that goes below this value will reset itself to its starting position.
    - **camera** is a pointer to the current SRE camera in the scene.
    - **playerObject** is a shared pointer to the GameObject that the last added CharacterController component is attached to.
- **LevelBuilder** sets the events from its argument to varibles for later use.
- **BuildFromFile** returns a level build by LoadLevel.
- **LoadLevel** creates a level, creates a GameObject and adds a CameraComponent to this GameObject, sets this GameObject as the levels cameraObject. Gets a reference to SRE camera from this GameObject with the CameraComponent::Mutator::GetCamera method and set a reference to this camera as the level's camera. Sets the CameraComponents updateEvent with the Behaviour::Mutator::SetUpdateEvent method. It then reads the json filename sent as arguement and creates all the needed GameObjects and add them to the levels levelObjects vector. For each GameObject it runs the method MakeGameObject with the tileID of that GameObject. The created level is then returned.
- **MakeGameObject** is returns a GameObject with the required components based on the tileID send as the argument. It checks if the tile json file has the following members and will and the written component if it does:
    - "sprite" SpriteRenderer
    - "physics" PhysicsBody
    - "characterController" CharacterController
    - "leveloverbox" LevelOverOnHitByCharacter

GameObject

The GameObject is a class that is used as a way of collecting components. These then describe the GameObject's functionality.

- **IsActive** this is used to check if the GameObject is active.
- **ToggleActive** this toggles the isActive variable so it can be turned on and off.
- **AddComponent** this function is used to add a component to the GameObject using its type
- **GetComponents** gets a vector of components of the given type.
- **GetTransform** gets the transform for the object

- **GetChildren** gets returns the children objects for the GameObject.
- **InitialiseSelf** used to call the OnInitialiseSelf event for the GameObject and the InitialiseSelf method for all its children
- **Initialise** used to call the OnInitialise event for the GameObject and calling the Initialise method for all its children

Component

Base class for all components.

- **GetOwner** return the GameObject the component is attached to.

Transform

Transform is inherits from Component and keeps track of the GameObject's position, rotation and scale. Have getters and setters for all three. It is added automatically in the GameObject's constructor.

Behaviour

Behaviour inherits from Component and handles the OnEnable and OnDisable calls from the GameObject for the components for the updateEvent. Have Friend class Mutator, an implementation of the Attorney-Client idiom (Bolton, 2006).

- **Behaviour** delegates the Enable and Disable methods to the OnEnable and OnDisable of its GameObject
- **Enable** subscribes the InitialiseSelf, Initialise, and Update methods to the responding methods from its GameObject, effectively making the Behaviour active.
- **Disable** unsubscribes the InitialiseSelf, Initialise, and Update methods to the responding methods from its GameObject, effectively making the Behaviour inactive.
- **Virtual**
    - InitialiseSelf, Initialise, and Update methods
- **Class Mutator** has friend class LevelBuilder.
    - **SetUpdateEvent** sets the arguments target Behaviour components updateEvent to the arguments updateEvent.

CharacterController

This component inherits from Behaviour. It is used to control the player character's movement and animation state (see below).

- **Update** this method is used to update the position for the physicsBody, and change the animation state.
- **InitialiseSelf** gets components, PhysicsBody, CharacterAnimatior, Transform, from its gameObject.
- **registerEvents** subscribes the keyEvent and for the physicsBody the OnCollisionEnter and OnCollisionExit event.
- **deregisterEvents** unsubscribes keyEvent and for the physicsBody the OnCollisionEnter and OnCollisionExit event.
- **OnKey** sets the right, left and jump variable based on player input.
- **OnCollisionEnter** checks if the game object has collided with something.
- **OnCollisionExit**
- **Class Mutator** has friend class LevelBuilder.
    - **SetKeyEvent** sets the arguments target CharacterControllers keyEvent to the arguments keyEvent.

CharacterAnimator

The character animator inherits Behaviour. It controls the animation of the player character.

- **enum class State** {Idle, WalkLeft, WalkRight, Jump, Air, Fall};
- **SetState** this sets the animation state for the character.
- **InitialiseSelf** this method gets the SpriteRenderer from the GameObject
- **Update** is used to update the animation based on clipTimer and state.
- **Class Mutator**
    - **AddSprite** adds a sprite to the sprites vector.

CameraComponent

CameraComponent inherits from Behaviour and operates the levels Camera to follow the levels playerObject. It has the class Mutator as a friend class.

- **CameraComponent** sets up an offset variable for use when controlling the camera based on the SRE window size.

- **Update** updates the cameras position and direction based on the playerObject's current position, with a slight offset in the x direction.
- **Class Mutator** has friend class LevelBuilder.
    - **GetCamera** returns the arguments target CameraComponent's SRE camera.
    - **SetHalfHeight** sets the arguments target CameraComponent's SRE camera's OrthographicProjection to the arguments value.
    - **SetPlayerTransform** sets the arguments target CameraComponent's playerObject variable to the arguments GameObject.

LevelOverOnHitByCharacter

A Behaviour used to end the level when the player character collides with a GameObject. Used to create the exit tile at the end of each level. Implements collision check by subscribing to the CollisionEnter event published by a PhysicsBody attached to the same GameObject.

LinkedList<T>

A template implementation of a linked list.

- **Template Parameter**:
    - **T**: The type of element stored in the list.
- **Methods**:
    - **GetLength**: Returns the number of elements stored in the list.
    - **Add**: Adds the supplied item to the list.
    - **RemoveAt**: Removes the item at the specified index.
    - **RemoveFirst**: Removes the first element found matching the supplied item
    - **RemoveAll**: Removes all elements found matching the supplied item
    - **Clear**: Removes all items currently in the list.
    - **Concatenate**: Appends the supplied list to the list.
- **Operator**:
    - **[]**: Indexer, returns the item at the supplied index.

Node

A node in the list. Used to implement recursive traversal.

- **Methods**:

  - **removeAt**: Returns the node which should be the calling Node’s next Node in order to remove the element at the supplied index.
  - **removeFirst**: Returns the node which should be the calling Node’s next Node in order to remove the first element matching the supplied item.
  - **atIndex**: Returns the Node which stores the element at the index supplied.

Delegate<R(Args…)>

A generic wrapper for function pointers. Implementation based on The Impossibly Fast C++ Delegates (Ryazanov, 2005). It does this by storing a pointer to one of four static functions which can each run a different type of function (instance method, const instance method, global function, lambda expression) and a pointer to the invocation target, which is used by the static functions.

- **Template Parameters**:
  - **R**: The return type of the stored function.
  - **Agrs…**: A parameter pack, of any size of types of arguments to supply to the stored function.
- **Factory Methods**:
  - **Create<T, *TMethod>**: Creates a Delegate for an instance method.
  - **Create<T,*TMethod> const**: Variant of above for const instance methods.
  - **Create<*TMethod>**: Creates a Delegate for a global function.
  - **Create**: Creates a Delegate for the supplied lambda expression.
- **Operators**:
  - **=**: Copies supplied Delegate’s data into this one.
  - **()**: Calls the stored function, with appropriate arguments passed. Returns any return value returned by the stored function.
  - **==**: Assesses equality of Delegates by comparing all data stored by both.
  - **!=**: Assesses inequality of Delegates by inverting **==**.

MulticastDelegate<R(Args…)>

A collection of delegates which can be invoked together. Implemented as an inheritor of LinkedList<Delegate>.

- **Template Parameters**:
  - **R**: The return type of stored Delegates.

    - **Args…**: A parameter pack, of any size of types of arguments to supply to the stored Delegates.
- **Methods**:
    - **Subscribe**: Adds the Supplied Delegate to this MulticastDelegate.
    - **Unsubscribe**: Removes the supplied Delegate from this MulticastDelegate. Note that this is not possible for lambda expression based Delegates, as C++ compilers create a new type for each lambda expression, regardless of signature. (cppreference.com, 2017)
    - **Invoke**: Calls all stored Delegates. Returns the returned value from the last Delegate called.

Event<R(Args…)>

A simpler, passable interface for MulticastDelegate. Allows a whole list of Delegates to be called as a single function call.

- **Template Parameters**:
    - **R**: The return type of this Event.
    - **Args…**: A parameter pack, of any size of types of arguments to supply to this Event.
- **Operators**:
    - **()**: Calls Invoke on the stored MulticastDelegate, passing supplied arguments and returning the return value of the call.
    - **+=**: Subscribes the supplied Delegate to the stored MulticastDelegate.
    - **-=**: Unsubscribes the supplied Delegate from the stored MulticastDelegate. See notes in MulticastDelegate<R(Args…)> for limitations.

JSON

Level

The file that is used to load the level, contains four different elements:

- **Id:** The id is a unique identifier for the level.
- **Name**: Also a identifier but not unique.
- **Array of tiles**: This is used to define the layout of the level. This is done by defining a array containing arrays of three elements. The first element of the array is the ID of a

tile, the second element is the X position of the tile, and the third element is the Y position of the tile.

- **Death level in Y**: This element is used to set the y position of the level where the player will be considered out of bounds and respawned at his initial position, that was set when the level was started.

Tile

The tile json is used to define properties of a GameObject in the game. This is done using four elements:

- **Id:** The id is a unique identifier for the tile.
- **Name:** Also a non-unique identifier.
- **Sprite:** Defines what sprite is used when the object is rendered, also means that during object creation the GameObject will get the component sprite renderer.
- **Physics:** This element is used to define the physics characteristics of the object. This is done using three sub-elements:
    - **Type:** This sets the physics body type, it can be static or dynamic.
    - **Sensor:** This is used to define if the physics component will be used as a sensor, is a boolean value.
    - **Density:** This describes the density of the physics object, the value is numeric.
- **CharacterController:** If this element exist and is set to one this tile will be receive the character controller component.

Sprite Atlas

This file is used to define what sprite is extracted from the sprite atlas image. The file contains an array of frames that is used to define what part of the image is a sprite. The objects in the frame has this structure:
Filename: This is the name of the file

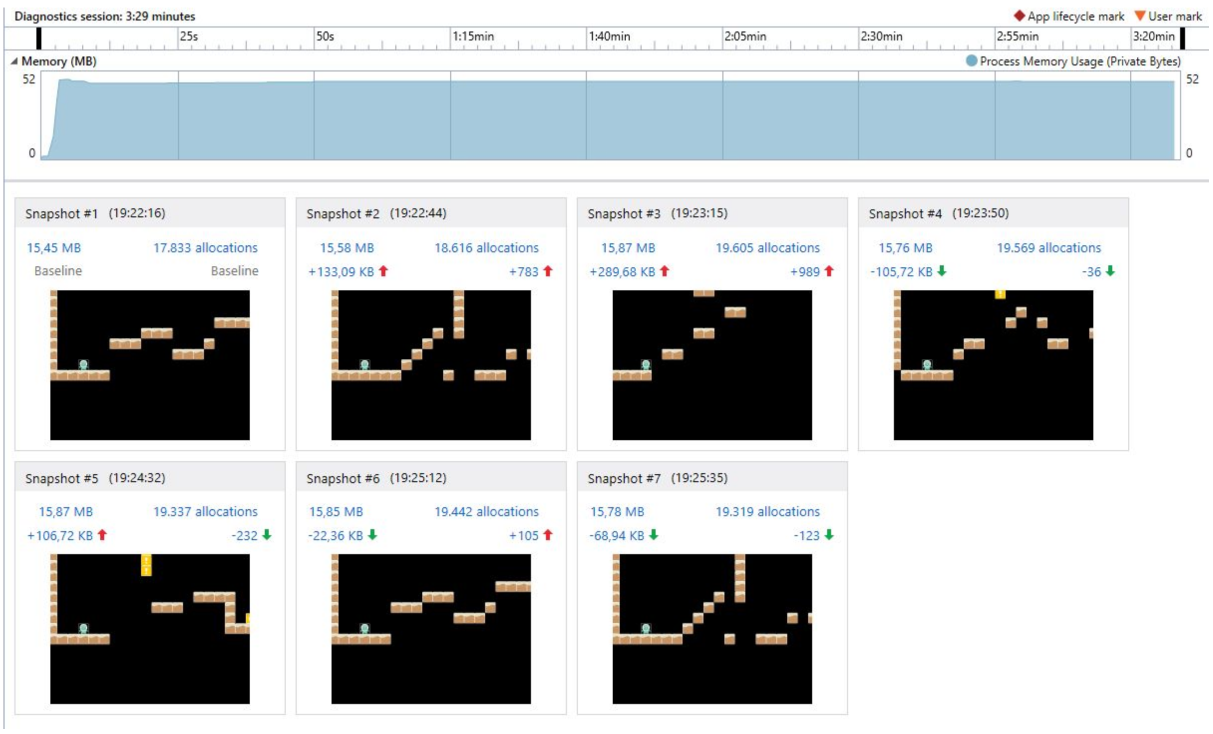
- **Frame**: This is used to find the position of the sprite in the image file, this is also where you set the width and height of the sprite.
- **Rotated:** Describes if the sprite is rotated
- **Trimmed:** Describes if the sprite is trimmed
- **Sprite source size:** This is used to describe the total size of the image file
- **Pivot:** This describes the pivot of the sprite

It can also contain contain metadata, under the key “meta”. This is used to describe general features of the image file, such as:

- **App:** Describes what app is used to create the JSON file
- **Version:** Describes what version of the app that was used
- **Image:** This is the name of the image that the JSON file refers to
- **Format:** Describes the image's color space of the image
- **Size:** Describes the width and the height of the image
- **Scale:** Describes the scale of the image
- **Smart update:** This is a hashvalue that is used by the texturepacker so it knows if it needs to rebuild the JSON-file, this is done to speed up the batching process.

Performance

Playing through the game provided a fairly consistent picture of memory usage, and no real performance issues have been observed. Snapshots done at the beginning of every played level on a playthrough completing level 2,3,4 once and level 1 two times are shown below.



Snapshot #1 (19:22:16)
15,45 MB 17.833 allocations
Baseline Baseline

Snapshot #2 (19:22:44)
15,58 MB 18.616 allocations
+133,09 KB ↑ +783 ↑

Snapshot #3 (19:23:15)
15,87 MB 19.605 allocations
+289,68 KB ↑ +989 ↑

Snapshot #4 (19:23:50)
15,76 MB 19.569 allocations
-105,72 KB ↓ -36 ↓

Snapshot #5 (19:24:32)
15,87 MB 19.337 allocations
+106,72 KB ↑ -232 ↓

Snapshot #6 (19:25:12)
15,85 MB 19.442 allocations
-22,36 KB ↓ +105 ↑

Snapshot #7 (19:25:35)
15,78 MB 19.319 allocations
-68,94 KB ↓ -123 ↓

Figure 5: *Memory use snapshots*

Problems if scaled

There exist a few possible performance problems if the levels the game engine created would scale massively. Currently the LevelBuilder creates each individual tile with no regards to what has been created before. Meaning it will read the tile from the hard drive every time it is created. If the level size were increased be a huge amount, this constant reading would slow down the LevelBuilder factory, making the creation of a new level take longer.

Another problem that exists if the size of the levels were increased massively the increase in update calls could potentially lead to slow down. The reason for this is that there is no check if a update for an object is required, e.g. GameObjects not in view do not necessarily require updates. This could be solved by sleeping objects, that are not on the screen or too far away for them to effect the game.

A scaling problem that would contribute to the previously mentioned problems is that every tile in the game is its own object. A perfect example for this would for instance be the ground and wall at the start of every level. A solution to this would be when generating the level to check if it is not possible to build larger objects of similar complexity so that the number of objects and component in the game would be reduced.

Final product

The flexibility of the game and the possibility for players to implement their own level into our game engine is a really positive aspect of the final product. It makes it possible for people not familiar with C++ to experiment with creating their own levels and puzzles. This is enhanced by the ability of the CharacterController to wall jump thereby adding more depth to the controls. If the ability to wall jump was not wanted in the game engine, a raycast from the position of the Player and down when making contact could be implemented to make sure the contact did not happen to the side.

The GameObject Component system mimics how most people would think about the different elements of a 2D platformer game. It “Allow a single entity to span multiple domains without coupling the domains to each other.“ (Robert Nystrom, 2014) Meaning it's easy to understand and work without adding much to the computational implementation.

The implemented event systems makes sure there are no unnecessary checks for whether or not an object needs to run behaviour in the game engines different update loops. The system provides a flexible base for implementing new behaviour, by only subscribing to what you need to make it work. The implemented OnEnable and OnDisable subscription system makes it easy to enable/disable every component on a GameObject one centralized place.

Collision could also be improved. Instead of just sending infor on the GameObject that have collided, more detailed info relating to the actual collision itself could provide more fine-grained collision based behaviour.

The current implementation is lacking some features that users might expect from this type of game, 2D platformer. Firstly we have not implemented sound effects in the game, this is something that would increase player enjoyment. Depending on the sound effect this could be added to the CharacterController for sound effects like running and jumping, alternatively this could its own component, so it could be added to other agents in the game. It could also be another element attached to a tile object in the game, where the sound effect would play when something interacted with the tile through collision. Secondly there is no music in the game. Music would probably be implemented on a level basis. Where the level would be the object that decided what music to play, that way you could have musical variety in the different levels. The final great omission of our game is a background for the game. This is would also be implemented on a level basis, making it it possible to give the player a greater impression to be moving through different environments.

Responsibilities

In our group shared responsibility for the project. So where certain group members would initiate certain elements, Joe did event system and components, Simon implemented scene and Scenemanager, Victor created the LevelBuilder and LevelManager. We all in the end would have to make changes and further develop each other's code.

References

cppreference.com (2017). *Lambda Expressions (Since C++11)*. [Online] Available at: http://en.cppreference.com/w/cpp/language/lambda (Accessed: 30 November 2017).

Bolton, A. R. (2006) *Friendship and the Attorney-Client Idiom*. [Online] Available at: http://www.drdobbs.com/friendship-and-the-attorney-client-idiom/184402053 (Accessed: 4 December 2017).

Ryazanov, S (2005). *The Impossibly Fast C++ Delegates*. [Online] Available at: https://www.codeproject.com/Articles/11015/The-Impossibly-Fast-C-Delegates (Accessed: 26 November 2017).

Nystrom, R. (2014) *Game Programming Patterns* [Online] Avaliable at: http://gameprogrammingpatterns.com/component.html (Accessed: 10 December 2017)