# Component-based game object system

Nicolas Porter

A dissertation submitted to

Carleton University

in partial fulfillment of the requirements for

COMP 4901

Dr. Doron Nussbaum

School of Computer Science

Carleton University

April 2012

## Table of Contents

## List of Figures

# 1 INTRODUCTION

## 1.1 High-level problem description

When someone tries to learn game development, they will usually try to make a relatively small game; one that has a modest game world which contains two or three game objects. The technique that is typically shown in beginner-level game development resources to implement such as game starts by instructing the developer to organize his game objects in a hierarchical manner. The developer then successfully implements the rest of his game. Feeling confident, he tries to add more game object types but quickly discovers that without resorting to ugly coding tricks, he would have to re-engineer his game object hierarchy. Although this works because his game is small, this technique of organizing game objects in a hierarchy is not compatible with today's game development process because of its inflexibility and maintenance problems which I will describe later in this paper. Furthermore, this system does not allow game object types to be defined in data. This means that the task of adding or removing game object types, as well as modifying a type's behaviors, is left to the software engineers.

A good game object system must be scalable such that game object types can be added easily to the system regardless of the number types already defined. It must also be maintainable such that behaviors can be added to or removed from a game object without problems. Lastly, it should be modular such that game object or their behaviors can be reused for different games.

The hierarchical game object system cannot support those requirements and thus, other approaches need to be explored. The candidate explored in this paper is the component-based game object system which uses composition, rather than inheritance, to define game object types and their behaviors.

## 1.2 Overview of Results

A hierarchical game object system is easy to understand, but scalability issues arise as the number of game object type increases. The component-based approach presents a solution to these issues, but it is much harder to implement in statically typed languages such as C++.

# 2 PREVIOUS WORKS

This chapter lists a few of the more popular component-based game object systems available.

## 2.1 Artemis

URL: `http://www.gamadu.com/artemis/`

Language: Java

The architecture of this system is very interesting. The game objects (referred as entities) are simply an id. They are aggregated in an entity manager which maps an entity to its components. The components only encapsulate state; they do not define a behavior. Instead, the behaviors of a game object are defined in what they call systems. A system acts on a subset of an entity's components. For example, the *MovementSystem* would act only on the *velocity* and *position* components of an entity. These systems are stored in what is called a system manager. Individual systems can be retrieved from the system manager. The communication between systems is done via function call.

## 2.2 Cistron

URL: `http://code.google.com/p/cistron/`

Language: C++

Cistron is lightweight, fast and flexible. It follows the component-based game object system described in this document fairly closely. Communication is done either by directly calling a method of a component, where the component is retrieved from the game object and the dynamically casted to the appropriate type, or via message passing. As for the state of the game object, it is encapsulated in the components as opposed to being shared as in my implementation.

## 2.3 Craftyjs

URL: `http://craftyjs.com/`

Language: Javascript

Because Javascript is a dynamic programming language, it lends itself really well to component-based game object systems. This particular implementation has game objects (referred as entities) which hold their state and components which encapsulate behaviors. Component to component communication is done via event callbacks.

## 3 SOME DEFINITIONS

Before we can talk about various ways to engineer a game object system, we must first define a few terms.

### 3.1 Game Objects

Game objects represent the actors of the game world. A game object can be either static or dynamic. Static game objects do not react to external stimuli and therefore, do not interact with the world or other game objects. They are often scenery elements, like a tree or a rock. Dynamic game objects, on the other hand, can interact with other game objects.

Arguably, the game camera could be considered a game object because while it does not affect the game world, it is affected by it. For example, a third-person camera needs to react when colliding with geometry in order to avoid clipping.

The Heads' Up Display (HUD) is not a game object since it is not part of the actual game world, it merely displays information about it. Similarly, post-processing effects are not considered game objects because they only affect the rasterized output and not the game world. A game object can have many behaviors, but it must have at least one otherwise it would not be considered a game object since it would be unable to interact with anything in the game world.

### 3.2 Game Object Behaviors

A behavior describes how a game object reacts to a certain internal or external stimuli. A reaction is the execution of the behavior and may or may not cause side effects in the game world.

### 3.3 Game Object Property

A game object property is a characteristic of a game object, such as its position or color.

### 3.4 Game World

The game world contains all game objects. Communication between game objects is performed at this level.

### 3.5 Game Object Model

The abstract representation of a game that defines the interactions between game objects, the game rules, etc.

### 3.6 Game Object Systems

The game object system is the part of the game engine that enables the implementation of the game object model. It describes the instantiation of game objects, how to manipulate a game object's state, how to define the behavior of a game object and how game objects communicate with each other.

## 3.7 Evaluation Criteria

Here is a list of features that I deem important in any good game object system:

### 3.7.1 Is it easy to add/modify/remove behaviors to/from a game object?

Drastic changes to the game play model can occur during the development of a game. Therefore, it should be easy to add or remove game object behaviors. The change should also be local to a game object, meaning that it should not require the modification of one or many game object types. Otherwise, it could incur significant maintenance issues. Consequently, the game object system should be designed in a way that does not require dependencies between the behaviors of a game object.

### 3.7.2 Can we reuse game object types, or their behaviors, in new games?

Game object types and especially behaviors should be as reusable as possible because it will reduce the development time of other games.

### 3.7.3 Is it easy to understand the code/architecture?

Game development teams are typically large, so many people will have to look at the game objects. Therefore, the code and design must be easy to understand.

### 3.7.4 Does it improve the efficiency of the game development process?

Can new game object types be built quickly? We also must factor in the development time of the game object system itself. Is it worth the effort?

### 3.7.5 Does the system scale?

In other words, how scalable is the system in terms of the number of game objects and the number of behaviors the game object type can define? In order to create rich interactive worlds, modern games typically define many game object types which have many behaviors. A good game object system will allow game developers to define a large number of game objects and behaviors.

### 3.7.6 Can this architecture be defined in terms of data?

Engineering time is not only expensive, but is also one of the bottlenecks of the game development process. Consequently, game developers create content authoring tools that allow artists, game designers and other non-developers to create content for the game. Not only that, it allows the artist to visualize and tweak his work while the game is running. These content authoring tools can be implemented very easily if most of the game's implementation is defined in data.

## 3.8 Hierarchical / Inheritance-based

In a hierarchical game object system, game object types are organized in a inheritance hierarchy. More generic game object types lie at the top of the hierarchy, while specialized ones lie at the bottom. The behaviors of a game object type are simply function definitions. This type of model lends itself well to object-oriented programming languages that support inheritance. It was commonly used during the 90's.

It is an easy and intuitive way to model game objects, because it leverages object-oriented concepts that virtually all developers are familiar with. Furthermore, it can be developed relatively quickly from scratch, since most languages used for game development support this system natively. This is why most online tutorials, geared towards beginner game programmers, will feature this architecture.

While it works well for games with simple rules and few game objects, this design does not scale. Firstly, in order to understand a game object type, you must understand its hierarchy. The developer must ensure that by modifying the derived type, he is not violating any assumptions made by parent types. As the number of game object types increases, the hierarchy deepens. It consequently becomes harder to understand a game object type. Furthermore, if a game object type is modified, then its derived types may have to to be modified as well. For example, if we have an *Enemy* interface, and we add the "runAway()" behavior to it, then we must implement that behavior in all game object types that implement the *Enemy* interface.

Another scalability problem arises when adding new behaviors to multiple game object types that are in different branches of the hierarchy. This issue is best demonstrated by example. Figure 3.1 shows example of a simple game object model implemented using a hierarchical game object system.

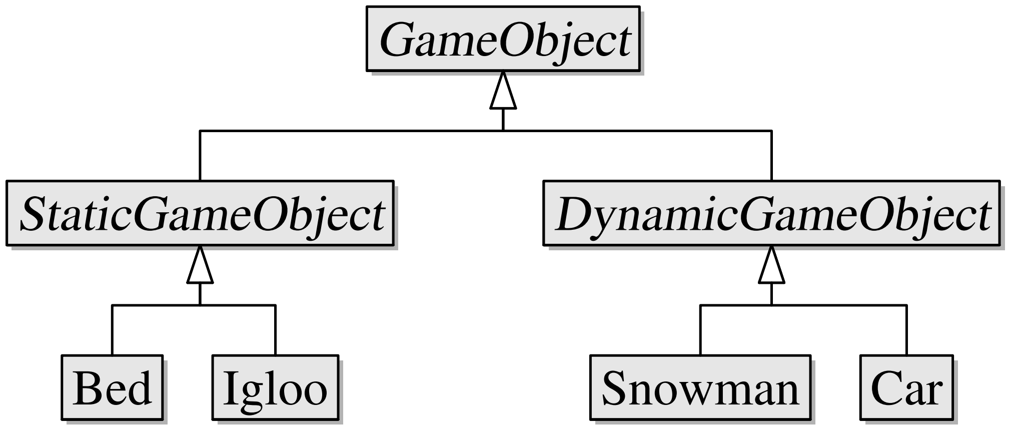


Figure 3.1: A simple game object hierarchy.

Let us pretend that during the development of the game, the “melt” capability needs to be added to the *Igloo* and *Snowman* types. We could copy the behavior in both types, as shown in Figure 3.2, but that approach is not maintainable because it violates the Don’t Repeat Yourself (DRY) principle.

Alternatively, we could use a mix-in class, as shown in Figure 3.3, but this limits us to a language that supports multiple inheritance, and poses yet another potential maintenance issue. We have already established that in order to understand a game object type, we must understand its hierarchy. But if we use multiple inheritance, the game object type is now part of multiple hierarchies. Therefore, if we want to modify our *Igloo* type, we now have to understand the *StaticGameObject* and *Meltable* hierarchies.

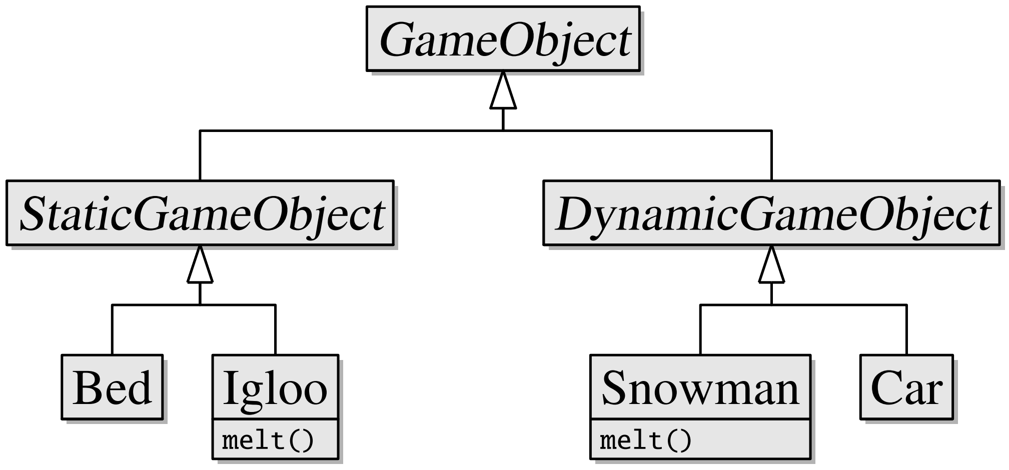


Figure 3.2: Adding the "melt()" behavior to two types that are in divergent branches.

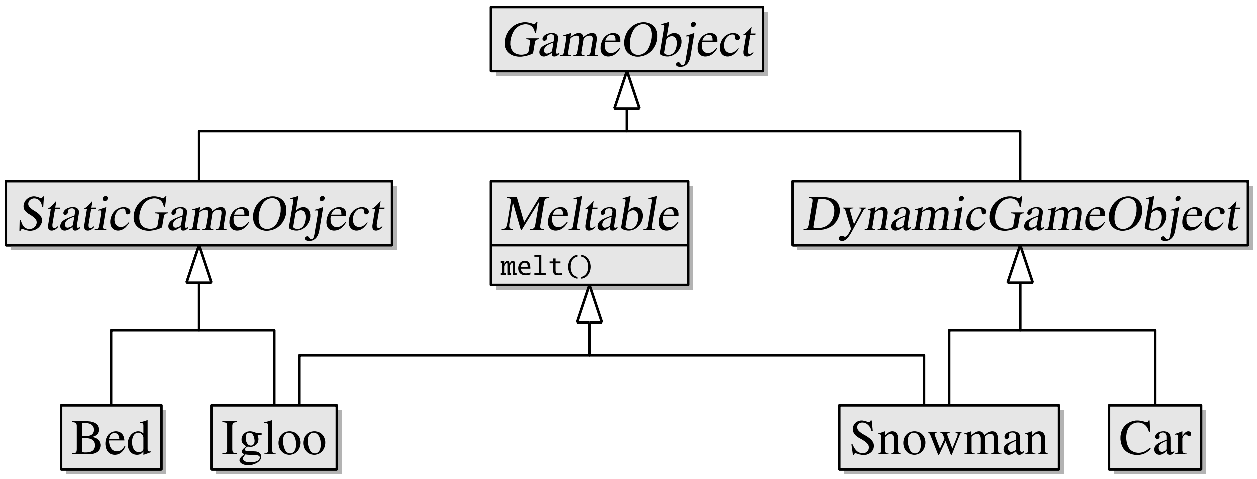


Figure 3.3: Implementing the "melt()" behavior with a mix-in.

What if the game design team suddenly decides that melting makes the game really competitive, and that all game objects should be meltable. How should we implement this new model? We have to move the "melt()" behavior upwards in the hierarchy, as shown by Figure 3.4. Alternatively, we could have all types inherit from our *Meltable* mix-in, but that requires us having to modify all existing types. It would not have been a bad approach if we would have implemented our model that way from the very beginning. We will explore this method in Hybrid part of this section.

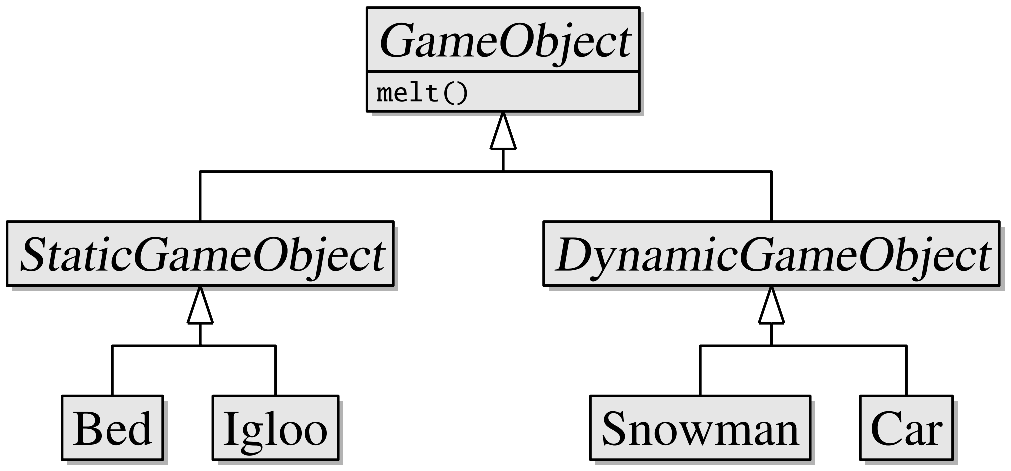


Figure 3.4: Floating the "melt()" behavior upwards.

After testing the new build of the game, the Quality Assurance (QA) team report that the player cannot save his game if he melts his *Bed*, as it is his only save point. But how do we remove the "melt()" behavior from our *Bed* type? The *GameObject* type requires us to implement it. We end up removing all logic in the *Bed* such that it is now blank. While that is not a bad solution, it is not ideal as it would be better if the behavior was completely removed.

Finally, this solution is not very modular. Since a game object type defines its own behaviors, it is tightly coupled with many of the engines' subsystems. Take some hypothetical "draw()" behavior, for example. It would most likely need to use the graphics subsystem, which means that a game object which implements that behavior will be coupled to the graphics subsystem.

This design does not allow the game objects to be defined in data. All the objects are hardcoded in this hierarchy. This means that software developers must be involved in the addition, modification or removal of game objects. Furthermore, in the case of a compiled language like C++, it means that the system must be recompiled every time a change is made.

If your game requires very few game object types, then this method is not so bad because it implies that the hierarchy is shallow. But for most games, this is not the case.

### 3.8.1 Pros and Cons

**Pros:**

- Simple to understand
- Can be developed quickly from scratch
- Statically typed
- Low overhead

**Cons:**

- Not flexible because it is hard to modify game object types after the hierarchy is created.
- Hard to maintain as the hierarchy becomes deeper.
- Game object may have behaviors they do not need, or should not have.

## 3.9 Component-Based

### 3.9.1 Description

In a component-based system, we flatten the game object hierarchy to a single base game object type which contains a list of behaviors called components. Since the behaviors are decoupled from game objects, the game object system not only becomes much more maintainable, but also very flexible. This system allows us to dynamically compose new game object types, which in turn makes it easy to define them in data. This allows non-developers to create content. One of the first uses of this technique was documented in the *Thief: The Dark Project* postmortem [12].

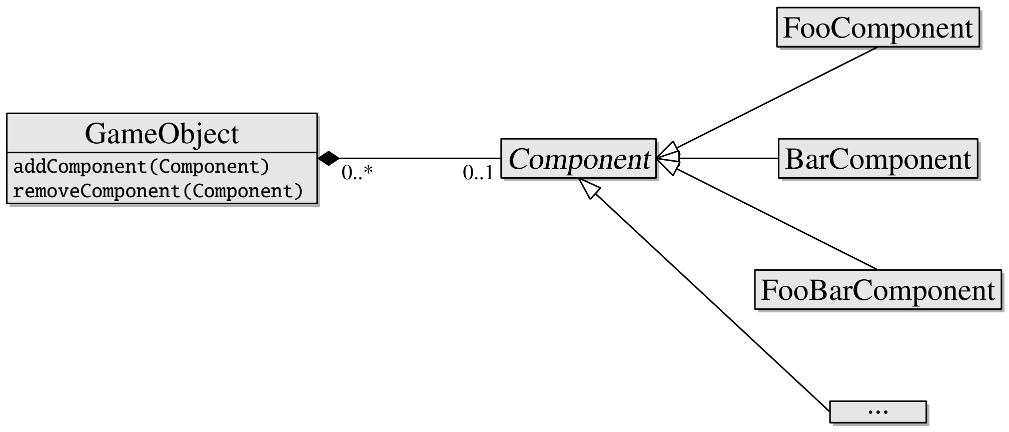


Figure 3.5: Game object are composed of many components.

The key to understand this system is to realize that we are no longer working in the *Object-oriented* paradigm. Game object types are no longer defined in the form of classes. Instead, their type is defined by the set of components they contain. These types are often described in data files and are then instantiated using factory functions. Alternatively, one could create a "prototype" instance and then clone it when another instance is needed.

Since a game object is composed of multiple parts, as shown in Figure **??**, we have to specify where game object's properties are actually stored. We can either store them in the game object itself, or spread them between the components. Both approaches have their advantages and disadvantages. In any case, it means that whoever holds the properties must expose an interface that will allow components to modify them.

In the case of a game object holding its properties, a property getter and setter must be added to its interface. This means that a component would need a way to retrieve some reference to the game object that owns it, but that is needed regardless due to the component messaging system (seen later). Because we define it dynamically, a game object does not know in advance which properties it will need, which means that there is a risk of property name collisions. For example, if you purchase two components from different manufacturers, and they both use an identical property name but they interpret it differently, your game object may not work properly. One simple solution would be to create a level of indirection between the game object's properties and the affected components. In a statically-typed language, the data structure backing the properties would need to be implemented as a tagged union. If you are using a dynamically-typed language, a simple associative array (map, dictionary, etc.) would work just fine.

If the properties are to be in held in the game object's components, then a component would need expose its property getters and setters. Then, in order to access or modify a property, a component would have to know which component holds the property. For example, if the *GraphicsComponent* needs the *position* property, it knows that it can get it from the *Movable* component. This approach is not very maintainable because it causes coupling between components. Alternatively, the game object could search each component, but that would be inefficient unless some elaborate caching mechanism is implemented. It is worth noting that while the *Artemis* framework employs this approach, it manages to works around these issues by making components pure data containers.

The game object interface allows components to be dynamically added/removed to/from game objects. This means that a component cannot make any assumptions about other components. Therefore, a component-based game object system must include some messaging subsystem that does not rely on components knowing about each other. A typical way to implement this is via message passing. For example, an *InputComponent* would want to send out a *shoot* message when the player presses the space bar. While this is a very scalable solution, it does make debugging harder. A common use case of the dynamic component addition and removal feature is that

While this method offers significant advantages over the traditional inheritance-based system, most notably in terms of scalability and maintainability. Unfortunately, it is not easy to develop such a system in a statically-typed language that does not support introspection, such as C++. However, when the system is fully implemented and a library of components is available, adding or maintaining game object types becomes very easy.

### 3.9.2 Pros and Cons

**Pros:**

- Scalable, in terms of number of game object types and number of behaviors per game object.
- Game object types can be defined in data.
- Modular, components can be reused between different games.
- After a library of components is assembled, it is easy to make new game object types.

**Cons:**

- Building the system takes a lot of time and effort.
- Dynamic typing.
- While the components themselves are easy to maintain, the component API is not flexible as changing it would require changing all components.

## 3.10 Hybrid

### 3.10.1 Description

We can also create a hybrid from the two types of systems. Instead of using factories to create game object types, behavior interfaces are statically embedded in a generic game object type, and concrete components are injected in the constructor, as shown in Figure 3.6. This is could be done by either derived types, or by constructing the game object directly.

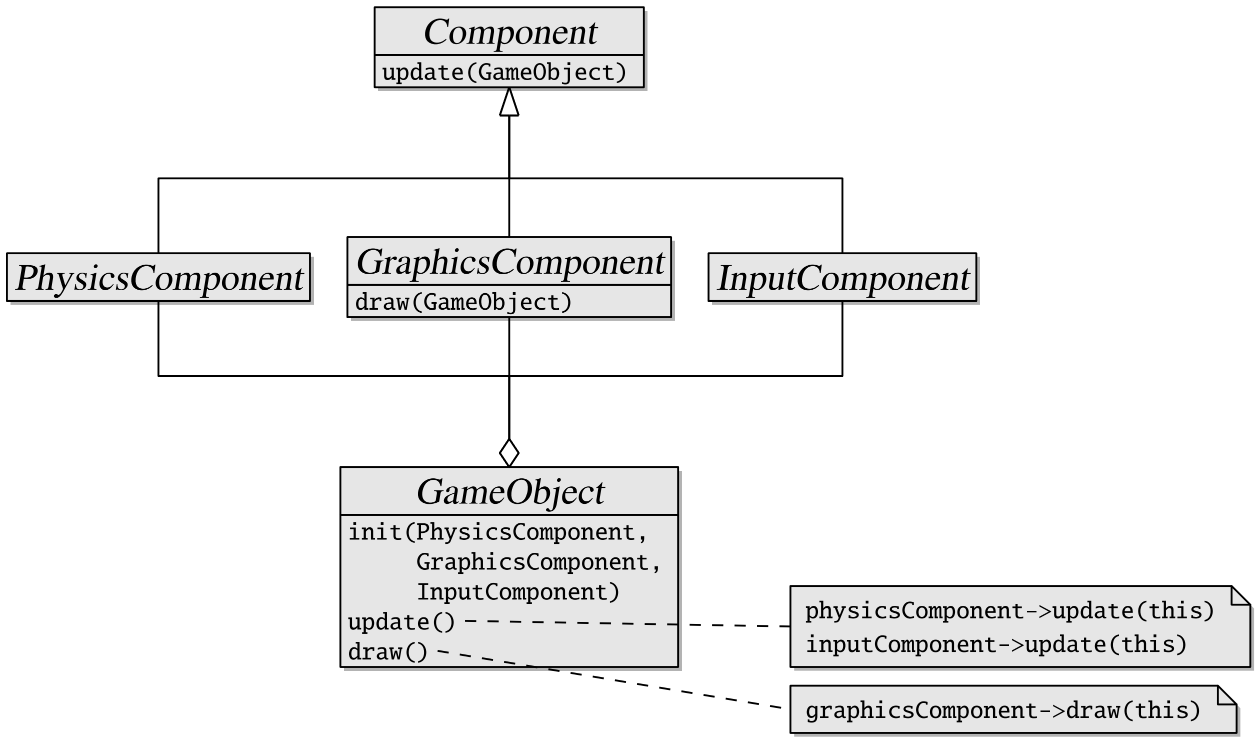
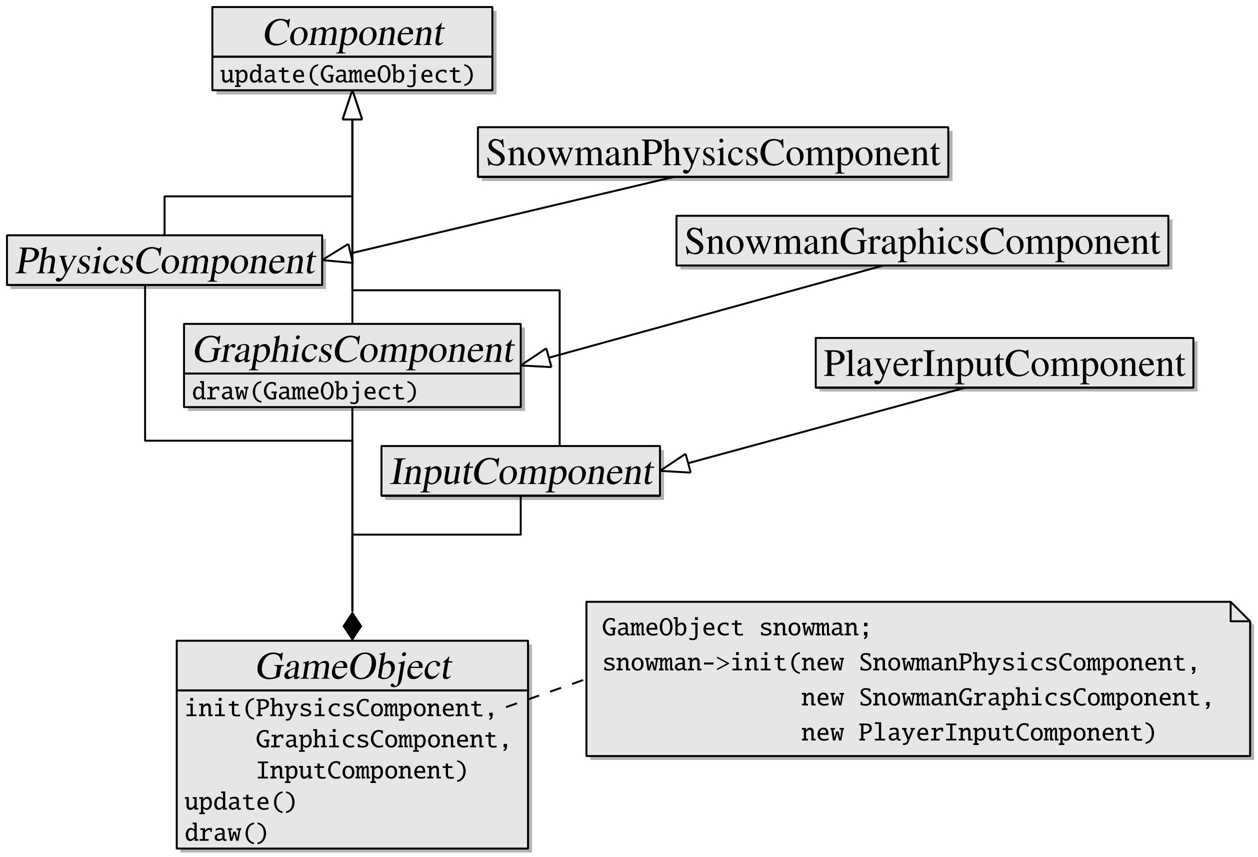


Figure 3.6: Hardcoded set of abstract component types in a game object.

Figure 3.7 shows how a simple, player-controlled *Snowman* would be implemented. Here, the component implementations would be injected in the game object when it is instantiated. Note that instanciating it this way means that there is no explicit *Snowman* type; the game object is only a *Snowman* by convention. If we wanted to enforce such type, we would need to create one that derives from *GameObject*, and injects the proper components on creation.



```
GameObject snowman;
snowman->init(new SnowmanPhysicsComponent,
              new SnowmanGraphicsComponent,
              new PlayerInputComponent)
```

Figure 3.7: Snowman game object implemented using components.

The problems with this approach is that we are loosing a lot of flexibility in terms of game object behaviors, since a class would only support a few select behavior types. That said, those types can be interfaces, which means that it would still allow a game object to change some of its behavior dynamically. It is an attractive solution if you are building a small game, and plan to reuse behaviors between games or you have a library of components already available.

### 3.10.2 Pros and Cons

**Pros:**

- Statically typed.
- Ability to change certain types of behaviors.
- Flexibility of components without the complexity of the system's implementation.

**Cons:**

- Similar to the ones outlined in the hierarchical system.

# 4 IMPLEMENTATION

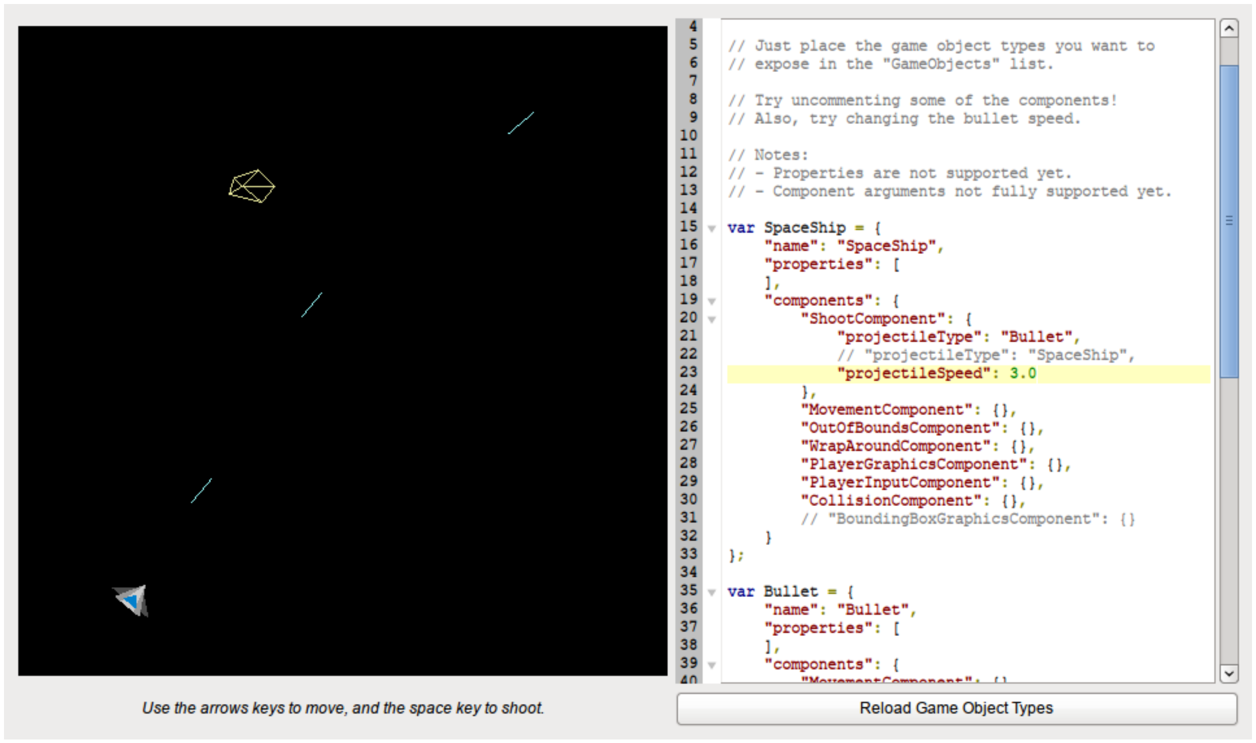
## 4.1 Instructions

Installation instructions can be found in the *README* file that was distributed with the implementation package.

## 4.2 Description

This paper also presents an implementation of a game object system. This implementation is written in C++, and is heavily reliant on the QT SDK (Software Development Kit). QT enables introspection of C++ objects through its dynamic property system. It does so by using a compiler, MOC (Meta-Object Compiler) which generates vanilla C++ code from C++ code that is annotated with special macros (the Q_OBJECT macro). It essentially turns C++ into a static/dynamic hybrid language.

In the component system implementation, game objects hold a list of components and a property map which is provided by QT. When a component is added to a game object, it injects the properties that it needs in the game object. However, care must be taken to avoid property name collisions, as there are no coping mechanisms. A rule of thumb is to inject a property only if you would normally have a data member for it in a game object hierarchy. For example, a *Movement* component would most likely have a *position* data member, so it should inject the *position* property. A *GraphicsComponent* may need to access the *position* property, but it shouldn't inject it in the game object.



```
// Just place the game object types you want to
// expose in the "GameObjects" list.

// Try uncommenting some of the components!
// Also, try changing the bullet speed.

// Notes:
// - Properties are not supported yet.
// - Component arguments not fully supported yet.

var SpaceShip = {
    "name": "SpaceShip",
    "properties": [
    ],
    "components": {
        "ShootComponent": {
            "projectileType": "Bullet",
            // "projectileType": "SpaceShip",
            "projectileSpeed": 3.0
        },
        "MovementComponent": {},
        "OutOfBoundsComponent": {},
        "WrapAroundComponent": {},
        "PlayerGraphicsComponent": {},
        "PlayerInputComponent": {},
        "CollisionComponent": {},
        // "BoundingBoxGraphicsComponent": {}
    }
};

var Bullet = {
    "name": "Bullet",
    "properties": [
    ],
    "components": {
```

Figure 4.1: A screen shot of a game that uses the component-based game object system implementation, as well as the component editor.

As Figure 4.1 shows, a sample game has been built with the component system. The implementation also features a component data file editor which enables you to dynamically modify game object types. The format of the game object type configuration file is in Javascript. This could allow interesting type generation, however that is not demonstrated in the sample configuration file provided with the implementation.

Interpreting this configuration file yields intermediary data structures called *descriptors*. There are two types of descriptors: a *ComponentDescriptor* and a *GameObjectDescriptor*.

The purpose of these structures is to decouple the encoding of the configuration file from the game engine's configuration interpreter. These descriptors are stored in*string* to *descriptor* maps that reside in *instanciators*. The purpose of the instanciators is to expose an API that allows the instanciation of an object from a string. In other words, they are factories. There are two types of *instanciators*: a *GameObjectInstantiator* and a *ComponentInstanciator*. The *GameObjectInstanciator* first creates a new game object, and then matches the *string* to a *GameObjectDescriptor*, which contains a list of *ComponentDescriptor* instances. A *ComponentInstanciator* is then used to these descriptors, and the resulting component instances are added to the game object.

The game object API allows components to be added or removed dynamically. Components hold an *owner* reference which points to the game object that contains it. Derived components will rarely use the reference directly since most of the game object API calls are wrapped by the abstract *Component* class, to reduce coupling and to increase maintainability.

Amongst other things, components are able to mark their owner for deletion. This is useful when you want to delete an object after a certain event occurs, for example when an object runs out of health and should disappear as a result. In the sample game provided with the implementation, this functionality is used when a collision occurs between an *Asteroid* and a *Bullet*.

The component communication system is implemented using message passing, where the game object is used as a message bus for component communication. A component registers certain messages upon insertion in a game object. This is just an optimization to prevent relaying messages to components that would not need to capture them. An example of message passing can be found in the *ShootComponent* and *PlayerInputComponent*. Basically, when the player presses the *space* key, the *PlayerInputComponent* fires a *shoot* event, which is the captured by the *ShootComponent*.

# 5 CONCLUSION

## 5.1 What was achieved

This paper analyzed and described the differences between hierarchical and component-based game object systems, and pointed out the pros and cons of those schemes.

It was concluded that component-based game engine systems are more maintainable and scalable than hierarchical ones. Furthermore, it allows the definition of game object types in terms of data, which is a desirable property of a game object system because it enables the development of content authoring tools.

It also presented an implementation of a component-based game object system, as well as a sample asteroids-like game that is built with it. The interface also features a simple game object type editor, to illustrate that game object types can indeed be defined in data and modified dynamically.

## 5.2 Future work

More work needs to be done to find a clean way to integrate components in other game engine subsystems. I believe that the approach employed by the *Artemis* framework is very close to an optimal solution. The behaviors (called *systems*) can be completely stateless, which means that more complex behaviors can be built by just composing systems.

The QT SDK's scripting capabilities, where Javascript code can be used to call native C++ code, should be explored. This feature is supposed to change drastically in QT's next version (5.0), which is why I did not use it for my implementation. It would allow me to implement components directly in data.

Another area that would be worth exploring is node-based visual programming user interfaces used for the development of game object types, component to component communication and even gameplay rules. Figure 5.1 shows an interface prototype that I was working for this project, but did not have a chance to complete.

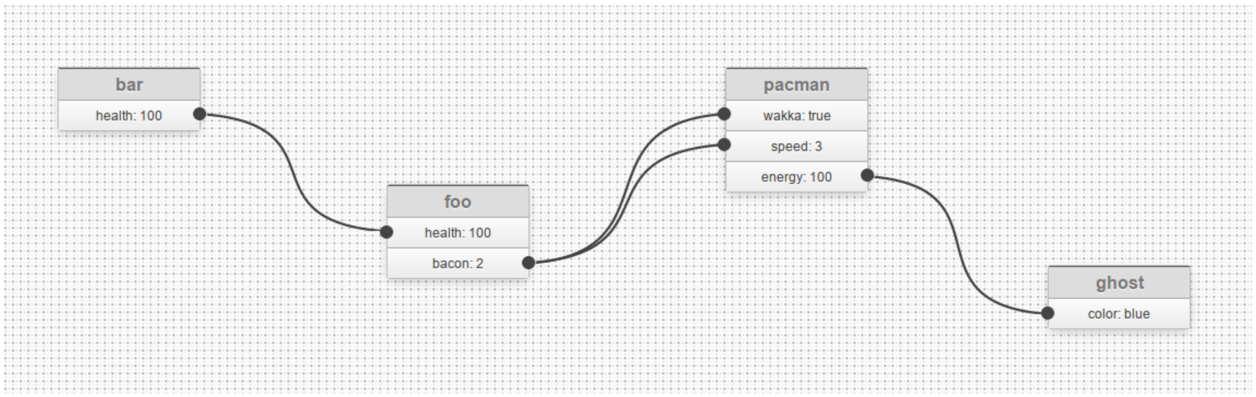


Figure 5.1: Prototype of a node-based visual programming interface for game object type creation.

## References

[1] Using the meta-object compiler (moc), . URL `http://qt-project.org/doc/qt-4.8/moc.html`.

[2] Introduction to game object system, . URL `http://www.neoaxis.com/wiki/Documentation/Articles/Introduction_to_Game_Object_System`. Retrieved February 27. 2012.

[3] Game objects, . URL `http://unity3d.com/support/documentation/Manual/GameObjects.html`. Retrieved February 21, 2012.

[4] What are the advantages and disadvantages to using a game engine?, July 2010. URL `http://gamedev.stackexchange.com/questions/859/what-are-the-advantages-and-disadvantages-to-using-a-game-engine`. Retrieved February 06, 2012.

[5] Adi. My proposal for a game engine architecture, April 2011. URL `http://www.adi-jurca.com/2011/04/18/my-proposal-for-a-game-engine-architecture/`. Retrieved February 21. 2012.

[6] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh McLoughlin. The case for research in game engine architecture. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 228–231, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-218-4. doi: 10.1145/1496984.1497031. URL `http://doi.acm.org/10.1145/1496984.1497031`.

[7] Arni Arent and Tiago Costa. Artemis entity system framework. URL `http://www.gamadu.com/artemis/`.

[8] Michael A. Carr-Robb-John. The game entity part i, ii, iii, iv, v, by, July 2011. URL `http://altdevblogaday.com/2011/07/10/the-game-entity-%E2%80%93-part-i-a-retrospect/`. Retrieved February 21, 2012.

[9] Karel Crombecq. Cistron. URL `http://code.google.com/p/cistron/`.

[10] GBGames. State of the art game objects, October 2010. URL `http://gbgames.com/blog/2010/10/state-of-the-art-game-objects/`. Retrieved February 21. 2012.

[11] J. Gregory. *Game Engine Architecture*. Ak Peters Series. A K Peters, 2009. ISBN 9781568814131. URL `http://books.google.ca/books?id=LJ20tsePKk4C`.

[12] Tom Leonard. Postmortem: Thief: The dark project, July 1999. URL `http://www.gamasutra.com/view/feature/3355/postmortem_thief_the_dark_project.php?print=1`.

[13] Gerold Meisinger. Why i switched from component-based game engine architecture to functional reactive programming, August 2010. URL `http://lambdor.net/?p=171`. Retrieved January 28, 2012.

[14] Jim Q. Ning. Ade - an architecture design environment for component-based software engineering. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 614–615, New York, NY, USA, 1997. ACM. ISBN 0-89791-914-9. doi: 10.1145/253228.253500. URL `http://doi.acm.org/10.1145/253228.253500`.

[15] Robert Nystrom. Game programming patterns - component, 2011. URL `http://gameprogrammingpatterns.com/component.html`.

[16] Pie21. Entity/component game design: A primer, July 2011. URL `http://piemaster.net/2011/07/entity-component-primer/`.

[17] Chris Stoy. Game object component system. In *Game Programming Gems 6*, pages 393–403. Charles River Media, 2006.