Technische Universität München
Fakultät für Informatik

Master's Thesis in Computational Science and Engineering

A quick prototyping tool for serious games with real time physics

Juan Haladjian

Technische Universität München
Fakultät für Informatik

Master's Thesis in Computational Science and Engineering

A quick prototyping tool for serious games with real time physics

Juan Haladjian

Date:
26.09.2011

Supervisor:
Prof. Bernd Brügge, Ph.D Advisor:

Dipl.-Inf. Univ. Damir Ismailović

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst habe und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 26.09.2011, ...
(Juan Haladjian)

Acknowledgments

I would like to express my gratitude to Prof. Bernd Brügge for making me part of his research team.

Special gratitude goes to my supervisor Damir for his great ideas, constant support and ability to motivate.

I am also pleased to acknowledge Barbara's tips on usability of my application and Jelena's corrections of this document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A case study is used to identify the two main factors that slow down the development speed of serious games with physics. First, the combination of different parameters needed for the physical simulation like coordinates, angles, physical values like friction and density, is hard to predict at compile time. This causes developers to repetitively test the application after every single parameter is modified.

Second, the interactive character of games makes their usability critical. A game with great features but minimal usability problems is abandoned before players can notice its great features [45]. Usability tests often cause changes in an advanced point in time, when the project is already complex.

A solution to the aforementioned problems is proposed, that uses prototypes for the creation of serious games with physics. The solution involves defining physical properties interactively with direct visual feedback. Coordinates, angles, scale factors, etc. can be directly set on a graphical interface, rather than coded.

Prototypes providing a good insight about the software can be created in around five minutes making *change* happen early in the project, as early as in the first meeting between developers and client. The prototyping tool is implemented on a tablet. Tablets' portability, multi-touch, and direct interaction capabilities promote collaboration between different stakeholders and help bridge their backgrounds' gap.

1.1 Outline

- **Chapter 2** provides background theory divided in two main sections: prototyping and serious games. Both sections first provide a general overview of the respective topics and then focus on specific theory that is needed for the comprehension of later chapters.

- **Chapter 3** describes a serious game with physics used as case study. An analysis of the game is used to identify the problems that affect the development of such games. Later chapters focus on a solution to these problems.

- **Chapter 4** provides a brief overview of applications and techniques that solve similar problems. It also describes how the related work directly influenced the objectives of this thesis. The rest of the chapter consists of scenario descriptions, requirements elicitation and initial analysis models. This Chapter contains the complete functional specification and therefore it can be seen as the Requirement Analysis Document (RAD).

- **Chapter 5** provides an overview of the software architecture and the design goals. The main part of this chapter documents the system design model with subsystem decomposition and persistent data management. This chapter represents the System Design Document (SDD).
- **Chapter 6** describes the detailed decomposition of subsystems into classes. Classes from the analysis model are described in more detail and additional classes are introduced. This chapter is organized by subsystems and is considered as the Object Design Document (ODD).
- **Chapter 7** describes in detail how the particle simulation is done, the techniques used for optimization, and the results achieved.
- **Chapter 8** presents an evaluation of the achieved functional and non-functional requirements after presenting the implemented prototype. The Chapter demonstrates the usage of the system in the creation of two different games with physics.
- **Chapter 9** shows the conclusion and the preview to future work.

Chapter 2

Theoretical Background

This chapter summarizes theory that supports the current thesis.

2.1 Software Prototyping

Software prototyping is similar to prototyping as known in other fields like manufacturing. It consists on creating incomplete versions of a software system with the purpose of testing it before starting with the creation of the real software. [16] defines prototypes as "an approximation of a product (or system) or its components in some form for a definite purpose in its implementation".

Prototyping has several benefits. It provides experts in the application domain new ideas and a better understanding of what the software should do. Experts in the solution domain gather experience in both, application and solution domain while developing the prototype, and then get valuable feedback from users and clients. Prototyping helps therefore identify and clarify requirements in early stages of the project. This is valuable, because often changes in a later state of the project imply considerably more work than creating a prototype. It also gives the development team a better idea of the amount of resources and time needed for the project.

Two main dimensions of prototypes have been identified [29]: *horizontal* and *vertical* prototypes. An horizontal prototype is a broad view of the entire system or subsystem. A common usage of horizontal prototypes is the development of GUI layouts where no GUI element (buttons, sliders, etc.) react to user interaction. A vertical prototype implements the whole functionality, including networking and persistency if needed, for a small part of the system. It is useful for eliciting in detail the requirements of a specific part of the system and to run performance measurements.

There exist different types of prototypes, most of which are based in two main families: *throwaway* and *evolutionary* prototypes. Throwaway prototypes are quick and cheap prototypes that get discarded rather than integrated in the final system. They are a cost effective way to refine requirements at early stages of a project. Examples of throwaway prototypes are paper prototypes and GUIs created with a GUI design tool. Evolutionary prototypes are more robust prototypes that are intended to evolve into (part of) the final system. A key fact behind evolutionary prototypes is that developers do not understand every requirement from the beginning, and implement only those they understand. An evolutionary prototype is iteratively refined and then given to clients and end users to experience it. In every iteration the functionality that is well understood by developers is integrated into the prototype [20].

Houde and Hill categorized prototypes as Role, Look and feel and Implementation prototypes [5], [11]. Role prototypes are used to define what an artifact does for a user (its purpose), without taking into account how this is achieved or how the interface will look like in the end. Look and feel prototypes are used to explore the different options how an artifact will be perceived by the user, and how this feels when interacting with it. Implementation prototypes try to determine whether an artifact can be constructed with existing technologies, what these technologies are and how can they be used.

2.1.1 Prototype Based Programming

Object oriented programming languages can be categorized in two big families: class based programming languages (CBL), and prototype based programming languages (PBL) [25]. CBL are based on the Aristotelian way of perceiving the world, where everything on earth is an instance of an ideal idea [42]. PBL are, on the other hand, based on Wittgenstein's philosophy where every object resembles some other object (the prototype) [42].

In CBL, *classes* define behavior and structure and *instances* are objects that carry the data and behave according to the class' specification. Behavior is shared by what is known as *inheritance*. In PBL, there are no classes and objects are created either from nothing (called *ex-nihilo* creation) or by *cloning* another existing object (the *prototype*). PBL that support ex-nihilo object creation require mechanisms to define structure and behavior. Cloned objects are initially identical to the prototype and can reuse its structure and behavior by what is known as *delegation*. Delegation is the process by which an object assigns tasks (behavior) to other objects. Objects typically keep a reference to the prototype they were cloned from, where structure and behavior are defined.

The main advantage of PBL over CBL is that no model is needed to create objects. Creating an object model that correctly represents different instances requires abstract thinking. Humans, however, prefer to think about concrete cases rather than about their abstractions. Furthermore, if requirements change in a software project after models have been defined, the models may not be valid anymore, which may cause big changes in the models.

The usage of prototype based programming languages is currently very limited. The most popular PBL is Javascript. Examples of CBL are the most popular languages like C, C++, Java, C#, etc. On one hand, developers are more familiar to CBL than to PBL. On the other, classes act as contracts; they guarantee that instances will be structured and behave in a certain way. This guarantee is not present in PBL, making developer mistakes not detectable at compile time, and as a consequence more error prone.

2.1.2 Domain Specific Language (DSL)

Domain Specific Languages (DSL) are programming languages with a high level abstract notation, used to solve problems in a specific domain. Their main advantage with respect to general purpose languages is their expressive power, which they achieve with special notations and abstractions [44]. Their main disadvantage is that this expressive power is only limited to a particular problem domain, which makes DSLs useless in slightly different domains than the ones they were created for. DSLs are typically worth employing when same high level semantics need to be used intensively. This is the case for example in database management systems, and that is the reason why SQL was created. High level procedures like creating a table or inserting new rows of data in an existing table involve many lower level operations, and are used frequently enough to justify the need of a DSL.

DSLs are typically declarative [44]. This means that what is programmed is *what* the program should do, without specifying *how* it has to be done. Some advantages of not letting developers specify how something should be done are usability, correctness and performance. Usability because the programmer can focus on defining his problem using high level semantics and without worrying about how it can be implemented. Users may not need to be experts in the solution domain in order to use a DSL. In fact, target users of many DSLs are clients for whom a software has been developed. The company who developed the software is usually the same that develops the DSL, targeting its own clients, so that they can monitor the running system. Safety and performance come from the simple fact that high level functionality provided by DSLs is often better in terms of correctness and performance than it could have been implemented by its users. If a developer could have created better functionality than that offered by the DSL then he would probably not have chosen to use it in the first place.

The Logo language is one of the first examples of a domain specific language. Although it is best known as a program to give instructions to a turtle, its scope is much wider. Some of its features include list data structures, recursion, boolean predicates and file handling. As its name suggests ("logos" means "world" in latin), it was designed to manipulate language, words and sentences [10] as opposed to other programming languages in the time of its creation that operated merely with numbers. Code sample 2.1 defines a function named `dash` with a parameter named `count`, that makes the turtle advance and draw the path it follows as a dashed line. The function body consist of repeating `count` times the procedure to make the turtle advance while drawing its path and advance without drawing its path. So when the function is called with a `count` parameter of 10, it will draw a line consisting on 10 dashes and 10 empty spaces, as shown in Figure 2.1.

Algorithm 2.1 Logo code example

```
to dash :count
   repeat :count [penup forward 4 pendown forward 4]
end

? clearscreen dash 10
```

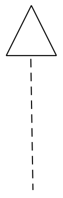


Figure 2.1: Logo graphics

[10] summarizes some advantages and disadvantages of DSL when compared to other general purpose languages, and states that good DSLs need to find a balance between both.

Advantages:

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
- DSLs can enhance productivity, reliability, maintainability, and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.

Disadvantages:

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users.
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.

2.2 Serious Games

2.2.1 Definition

Michael and Chen define *games* as a voluntary activity separated from real life that absorbs the player's full attention and that is played in a separate world with a well defined set of rules [27]. Schell adds a few elements to the definition: *goals* the player tries to achieve, *conflicts* that make the goals hard for the player to achieve and *disequilibrial outcome*, or in other words, games can be won or lost [36].

Although "*having fun*" is the primary reason for playing games, none of the previous definitions included it as an element or part of a game. This is because "having fun" is a consequence of playing a game, and not part of it [27]. Schell addresses the fun element by defining games as a problem solving activity that humans enjoy because of an evolved survival mechanism. By playing games, we solve problems, and by having fun, we play more games, so we solve more problems. Solving problems is our biggest advantage from other species, so solving problems makes us stronger and more likely to survive. He argues that even "War", a game with a completely random outcome involves problem solving. In "War", two players have a stack of numbered cards. Both players flip over at the same time the top card of the stack. The player with the highest numbered card keeps his card and earns the other player's card. The player that runs out of card looses. Although the outcome of the game can not be controlled, children playing this game are not aware or at least not convinced of this. The problem they try to solve is controlling fate, and they even try different ways of flipping over cards on the table, trying to influence the outcome of the game. When they realize they can not control fate, they stop playing it, because it stops being a *game* for them, and starts being an *activity.*

Corti states: "Games are nothing more than a vividly recreated environment and/or system in which the user has to solve a problem or series of problems. Solving that problem, be it 'how to kill 100 aliens as fast as possible without dying yourself' or 'how to settle a contractual dispute with a fictional client', is what derives satisfaction on the learner's part" [4].

Most authors agree that *serious games* "are games that do not have entertainment, enjoyment or fun as their primary purpose",[27] [41]. Note that this definition only states that fun is not the main purpose, but does not negate the existence of fun in serious games. Serious games typically have the purpose of developing knowledge, like learning to manage a big company and improving skills, like learning to drive a vehicle. The reason why people might want to play a serious game instead of developing these skills naturally in real life is that the real life experience might be too risky, too costly, or even physically impossible to achieve [4]. Some common application fields of serious games include education, military, city planning, emergency management, etc.

"Triage Trainer, a healthcare training simulation" is a good example of a risky and costly real life situation that is therefore only practicable in a digital form. The game is an emergency management serious game that targets professionals in the healthcare field, like doctors, nurses and paramedics [27]. It starts with an explosion in a crowded street, where players are called to and asked to assess the degree of injuries of the different patients. Game data such as injuries, their visual representation and their

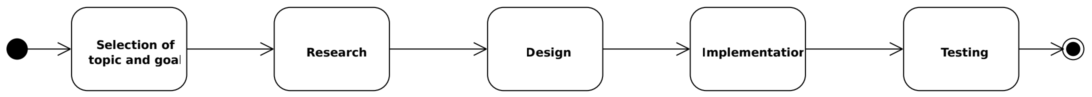


Figure 2.2: Game design phases [45].

evolution over time comes from experts in the field. The best possible diagnose is also part of the game's knowledge, in order to rate player's assessment quality. Thanks to this game, players are able to put into practice their medical knowledge and to use standard medical protocols at the same time.

Another relevant category of serious games is the advergaming. Advergames do not have the purpose of teaching concepts or skills. They are used for marketing and advertising purposes, to make a new brand known, or a known brand more popular, to attract users to a website, etc. In advergames, the advertising message can be integrated into the gameplay, or it may contain separate elements that provide the message, like in the case of Super Monkey ball, where bananas with the Dole brand had to be collected by the player [43].

The big power of serious games resides in their ability to captivate players [18]. It can be proven neurophysiologically that our body constantly seeks pleasant activities, and that when an activity is pleasant, we will try to repeat it. Dopamine is a hormone animals produce during and after pleasant activities. [31] conducted an experiment, where mice were subjected to run away tasks with a sweet reward for reaching the goal point, after their dopamine levels were increased by 70%. They observed that the mice had a bigger reward incentive, causing them to distract less often and to move more directly to the goal. The experiment suggests that dopamine (and therefore pleasant activities) increases animal desire for pleasure.

It is a common belief that games help directly or indirectly develop analytical, spatial, strategic, recollection and psychomotor skills. Effects of serious games are also considered to be positive in general. Serious games promote learning [41]. Besides motivating players, as described previously, they help introduce new concepts by supporting them with a story and by engaging players psychologically [4], making the concepts more likely to be understood and remembered by the player. Repeatability is another strength of serious games. If a player cannot reach a goal, he can try again and again until he achieves the necessary skills that allow him to reach it [4].

2.2.2 Game Engineering

Chris Crawford [45] describes the game engineering process as a sequence of phases. Because game engineering is a concrete case of software engineering, the phases in the creation of the game are very similar to the phases in the creation of a software. The main phases the author identifies are: selection of topic and goal, research, design, implementation, and testing. These are shown as an activity diagram with UML notation in Figure 2.2. In this section brief summary of these phases is presented based mainly in [45].

What every game has (or should have) is a clearly defined **goal and a topic**. The goal is what is intended that the player experiences when playing the game. If the game is educational, the goal is that the player learns concrete topics. The topic is the skin of the game, how the game expresses to the player. Although the topic of the game is often the first to be identified, this should be subordinated to the goals. The goals should be the strongest force in a game design, and even the topics of the game can be modified for the sake of achieving the goals.

Research and preparation is the second phase. It is important for the game engineer to understand the environment of the game. If the game is based on a historical civilization, the historical context of the civilization, its customs and way of living should be known. But this phase is not only limited to the topic of the game. The developers already get implementation ideas during the research phase, and are able to assess whether they can be implemented or not. The different ideas should then be contrasted, because often they will contradict each other.

Design is probably the most important part of the game lifecycle, and it is also the hardest. According to Brügge et al [2], design is hard because it focuses on the solution domain, which changes rapidly. For example, new technologies will often appear before the software is productive. A good design has to be able to cope with these issues and this is achieved by decomposing the system into components that are as much cohesive and decoupled from each other as possible. Crawford proposes a decomposition not of the system, but of the design methodology, in three parts: I/O, game structure and program structure.

In the I/O phase the input and output devices and features are decided. Output is currently mainly restricted to image and sound. While selection of visualization techniques may require a lot of effort and even the research of state of the art algorithms, sound requires little or no design. The I/O design phase is very important because it determines what the player will see and how he will feel when playing the game. If the game has a lot of functionality and great structure but a poorly designed I/O that affects its usability, players will leave the game before they can notice anything about its functionality or structure. Games with a lot of I/O give players more freedom to express what they need from the game, but too many ways of interacting with the game makes the game less usable, so a good tradeoff must be found.

The game structure phase consists of identifying and describing the main representative elements of the game. These elements should represent the issues addressed in the game, and should be understandable. Some examples of common representative elements in a game are moving and shooting. The representative elements should be manipulable in the sense that the player should be able to adapt this element to their needs. For example, if the chosen element is shooting, the game should not only allow shooting enemies at the head when they show up. The player should be able to decide when and how to shoot. Afterwards and based on these representative elements, features are progressively added to the game design. Some very complex features may be almost completely ignored by users, while very simple features may be great ideas that the players will like [3]. Finding the last ones and deciding which complex features to leave out is done in this phase. Performance will also suffer from adding too many features to the game, so it should be taken into account when deciding what features to keep.

The program structure puts the I/O and the game structure together in a real product. It focuses on the technical aspects of the design, that will directly influence the game's implementation. Some important aspects to consider in this stage are memory mapping of the different game objects, definitions of critical variables and subroutines and the game flow.

After the design an evaluation phase takes place in which the game engineer has to decide whether to abort or to proceed with the creation of the game, based on the information gathered up to this phase. One thing that needs to be done is to analyze whether all three design structures described above are compatible with each other. Another is to contrast the design with the original goals of the game. A key question in this is whether the player will effectively experience what is intended by the game. Finally, stability of the game has to be analyzed to see if the game can somehow get out of control such that achieving goals is either trivial or impossible. This may be the case for example in a race game, where the scenario is made in such a way that the starting point and the end point are very close together in space, and the game has no way to check whether the player followed the path or not. The player can take the shortcut and win the game trivially. Another case is for example if a player can run out of money in a game where money is the only way to advance in the game, and the only way to get more money as well.

The author identifies a **pre-programming** phase where developers document, formalize and commit all the artifacts produced by the design. He also describes the implementation phase as trivial depending on how good the design has been done, but cumbersome and time consuming. However the implementation is not where projects fail. This happens either during design or testing.

The **testing** phase is where design and programming flaws have to be identified. Crawford categorizes flaws as non-fatal and fatal. Non-fatal flaws are usually caused by lacks or excesses. For example, there may be too many elements on the screen, or the game lacks action. Although these flaws may not be critical, the flaw should be deeply analyzed in order to find out the reasons that originate it. Sometimes a fatal flaw shows its symptoms as a non-fatal flaw. In the best case, the flaw can be repaired. This requires contrasting different options and carefully thinking how it will be repaired. A small bug fix in some part of a complex software may create many new bugs somewhere else. Fatal flaws are unforeseen conflicts between main game elements, product of a bad design. In this case, the implementation should be aborted, and design assumptions reconsidered.

The testing can be divided in two phases. First, the developers are the only ones who test the game with the purpose of fixing issues. Only in the end the game is given to users, as they may get contaminated by testing it beforehand, although this is in contrast to Itterheim [15], who says that games should be tested as early as possible with final users.

Testers should be familiar with playing games to be able to compare the game with other games they know in order to give critics and say what they find a good idea. It is also positive that players have experience in game design so they can share their knowledge and opinion about other technical issues. Knowing as much about the tester's background is important in order to appreciate when their suggestions are subjective.

Many critics obtained during this phase will not be applicable, because of project budget limitations, lack of memory for new the features, etc. Testing, polishing the game, and integrating the changes should be repeated as many times as possible, as even in that case, the game may still contain bugs.

The proposed working methodology summarized the main steps in the development of the game. Special importance is given to the design part because errors that originate from a poor or incomplete design can remain undetected until the testing phase. The solutions described in this section are not meant to be applied strictly in the design of every game, but to be adapted to every concrete problem, just like a design pattern.

2.2.3 Serious Game Engineering

Because serious games are still games, many game design considerations also apply to serious games. This section describes how they differ and what additional issues need to be addressed by serious game developers, based on the research of Michael and Chen [27].

While video games are typically made for the latest hardware, serious games need to run on older and heterogeneous systems. In many schools for example, the budget can afford hardware that is far from being the latest, and different schools use different hardware and operating systems. Serious games need to be designed taking this into account.

Serious games target a wider range of users, some of them with no experience on gaming, like educators, corporate executives, etc. This poses some difficulties to serious game developers because they can make less assumptions about the user's knowledge and expectations.

As stated in [35], a simulation can never contain every possible aspect of the phenomena being simulated. For the development of a serious game it needs to be chosen what to simulate and what to leave out. A very realistic simulation can go down to the molecular level, and even further. But the costs of such an accurate simulation require a supercomputer to simulate a single drop of water. A model needs to be found that efficiently reproduces world laws up to a certain accuracy.

Simplifications usually done in normal games may not apply in a serious game such as using random number generators, compressing time, assuming perfect communication between people, etc. The rules and models behind the simulation need to be chosen carefully in order to match reality, while preserving fun, if possible. A wrong assumption may lead to teaching the wrong skills to the learner.

Assessment of learners in the current education system is commonly done after the learner carried out an entire unit. One of the advantages of assessment in serious games is that it can be done constantly while playing. Furthermore, the assessment may adapt the game, for example in order to make the player focus on his weaknesses. In order to enable such a functionality, serious games need special models with special data structures.

Educative serious games are meant to assist teachers, not replace them. They should therefore provide additional functionality, like an observer mode where the teacher can not only see what the player sees but also modify the game's state in order to control the student's learning, or rate him. Coaching, pausing the game, giving homework and assisting teachers in correction are other examples of functionality needed by serious games.

While in traditional games "fun" is a requirement, it is not so clear that some serious games should be fun. Fun is a good way to motivate players to learn, and can for example be a valuable element in educational serious games for kids. But it in an emergency management serious game like the one described previously in this chapter, the fun character may distract players from their tasks. In games where fun is not applicable, there are other things that can motivate players, like the satisfaction of problem solving, as explained by Schell [36], or the feeling of being part of the game and the ability to identify and express ourselves through the game.

As a conclusion, the simulation of real world processes and effects has more relevance in the case of serious games and this needs to be addressed carefully by serious game engineers. Furthermore, serious games have other requirements, like the need to integrate the game to a class curriculum, or the need to assess the learning of players, which results in functionality like providing separate tasks or homework.

2.3 Multitouch Gestures

Multitouch gestures are finger movements on a touch screen used to interact with a device. There exist many of them. The ones that are relevant for this master thesis are shown in Table 2.1.

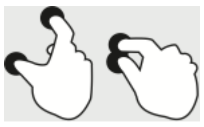
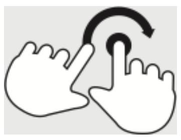
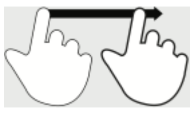
Name	Image	Description
Tap		Touch briefly the screen with the fingertip
Pinch		Touch screen with two fingers and bring them close together. Moving fingers away from each other is called *reverse pinch*.
Rotate		Touch screen with a finger and move the other finger in a clockwise or counter-clockwise direction. Another variant of the rotate gesture can be achieved by rotating both fingers (in the same direction) and not just one.
Swipe / Move		Move the fingertip over the screen without loosing contact.

Table 2.1: Most common multitouch gestures.

Chapter 3

Case study

The aim of this thesis is to improve the development process of games with real time physics. The difficulties that arise when developing such an application are analyzed in this chapter with a real case study.

The case study is an educational serious game that targets the iPad platform and teaches mathematics for preschool children. The topics taught in the game are counting, adding, subtracting and writing numbers. The game is composed of five mini-games and a bonus game. Each mini-game consists of specific tasks and exercises that need to be solved by the player. While every mini-game covers one or more educative topics, the bonus game has no educative character and serves the purpose of reward. The better the player has previously solved the mini-game exercises, the more fun the bonus game should be. This is achieved by awarding the player with stones in each mini-game, that are used as weapon to attack enemies in the bonus game.

The game takes place in a cave. The game characters, a bear and a penguin, appear in some way every mini-game and are used as instructors. They help the player understand what has to be done in every mini-game and give auditive hints when the player needs them. The initial screen (sometimes referred to as Intro game) is mainly a background image that lets the player select the mini-game he will play. This is done by tapping each of the six different stones (one for each mini-game). The different mini-games are:

- The Drawing Game, where the player must draw a number by touching the screen.

- The Bug Game, where there are bugs the player must smash or lead out of the cave after identifying whether they are good or bad by counting the amount of stains they have.

- The Labyrinth Game, where a goblin must be lead out of a labyrinth while collecting numbers, that will need to be added up before exiting the labyrinth.

- The Rope Game where a ball must be thrown in the bag labeled with the correct result for the mathematical operation shown on the ball.

- The Geometry Game where figures of different shape and color must be lured into the correct spot.
- The Bonus Game, a “Space Invaders” type of game where six goblins shoot stones to UFOs that move from the top to the bottom of the screen.

3.1 Development

A prototype of the game was developed within four months by a group of nine student developers, two project coordinators and two students working on the usability tests. The prototype includes an advanced version of every game with the exception of the Geometry and the Bonus games. The clients of the project were the author of the game series and two representatives of the editorial that published previous versions of the game.

The Scrum development methodology was used, although its guidelines were not strictly followed. Scrums did not happen daily but weekly. Every week developers met in the same place at the same time and reported their progress since the previous week, their objectives for the incoming week, and the problems that were blocking them. The project leader took the responsibility of integration of the different game components and delivered it monthly to the clients. After the delivery, the development team met the clients to validate requirements and get feedback.

Alpha tests of each mini-game were conducted by different members of the team. Beta tests were done by around five end users on the created prototypes.

3.2 The Rope Game

The *Rope Game* is one of the five mini-games, where the game characters hang from a rope in a cave holding a bag with a number. On the ground there is a mud ball with a simple addition or subtraction operation. The player has to solve the operation and throw the ball into the bag with the correct result. To throw the ball, the player touches the ball and quickly moves the finger in a direction before loosing contact. When the ball misses the bag, it splashes against the wall in the background, or it enters one of the goblin’s caves. A sketch of the game can be seen in Figure 3.1.

In the Rope Game, the game characters are not modeled as a simple sprite, but have instead rag doll physics. They are composed of different parts (arms, legs, torso, head) that are joined together. Every part behaves physically (for example, they are affected by gravity) and reacts to collisions against other objects and joints with other parts. When the ball is thrown at the bags, it either enters the bag, splashes on the background wall, or it enters a cave. Caves are modeled as polygons. In order to make the ball enter a cave, the collision between the ball (circle) and the cave (polygon) has to be detected. All these features and the rope simulation made it necessary to use a physics engine. The **Box2d** physics engine was used for this purpose, together with the Cocos2d Framework.

The `PhysicalObject` class wraps the different variables required by the physics engine like bodies, fixtures, etc. It is used as a base class for game objects like the

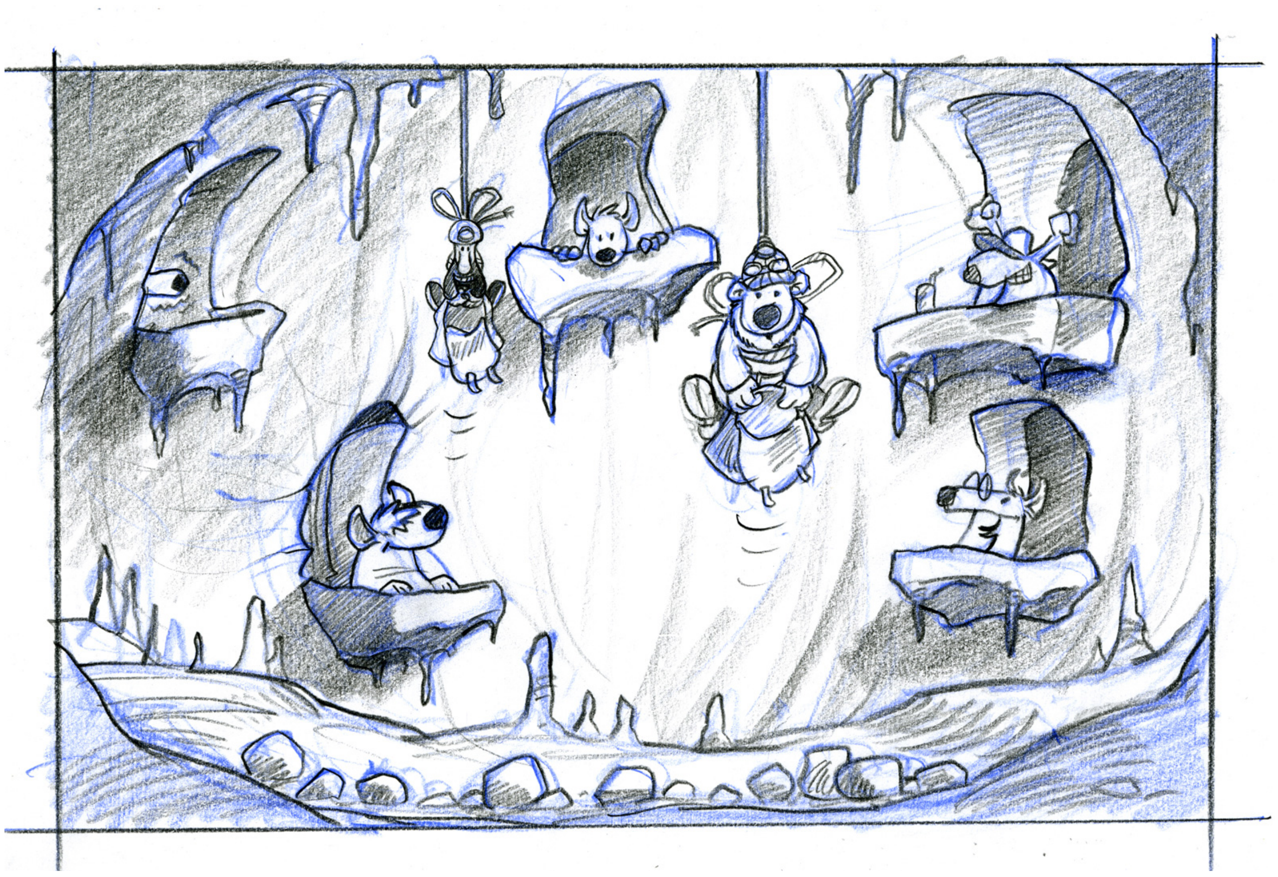


Figure 3.1: Sketch of the *Rope Game*

Algorithm 3.1 Update of an objet's sprite according to its body's (variable `b`) position and angle.

```
sprite.position = ccp(b->GetPosition().x * PTM_RATIO,
                      b->GetPosition().y * PTM_RATIO);
sprite.rotation = -1 * CC_RADIANS_TO_DEGREES(b->GetAngle());
```

`Goblin`, the `Cave` and the `Ball`. The `PhysicalObject` and `Shape` taxonomies are illustrated in Figure 3.2. The `Character` class is composed of six `PhysicalObjects`: five rectangles for the legs, arms and torso, and a circle for the bag.

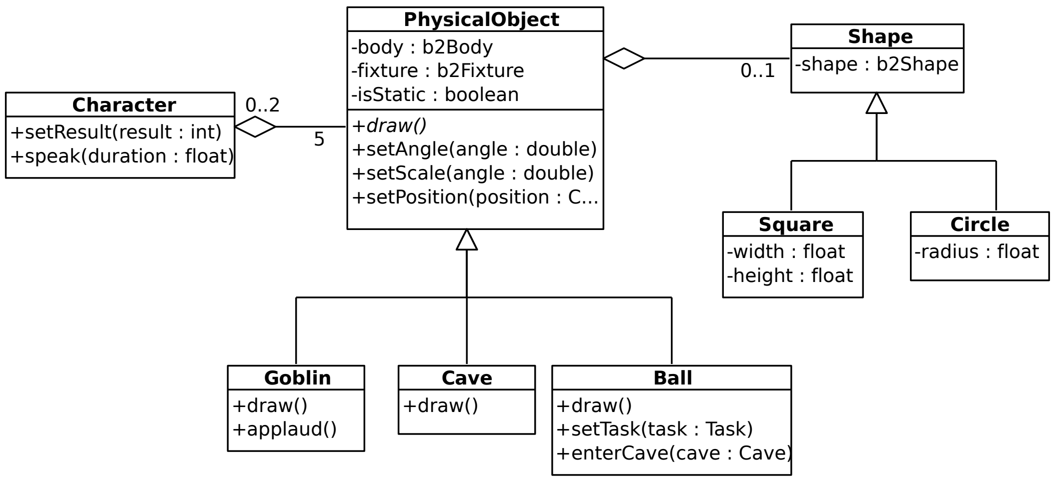


Figure 3.2: Structure of the *Rope Game*

The variables needed to define a `PhysicalObject` are a fixture, a body, and a sprite. The fixture is used to define object's properties such as density, friction and restitution. It also carries the information needed for collision detection and handling, such as the shape and the collision flags. The body is used to determine the kind of an object (static, dynamic, etc), its position and velocity and also implements functionality to apply impulses, torques and to exert forces on objects. Because Box2d has no rendering functionality, the sprite's position and angle has to be updated every frame according to values calculated in the simulation. Algorithm 3.1 shows how the update of the sprites is done with the data delivered by the physics engine.

3.3 Observed Problems

The development speed of the Rope Game was mainly limited by physics related issues and by changes in the requirements. Physics related problems were caused by the great difficulty to predict the behavior of the simulation at compile time. Changes in the requirements were mainly due to changes in the perception the clients had about the game after seeing a prototype. Usability issues were also detected when usability tests were conducted with end-users.

Algorithm 3.2 Character's left leg properties definition.

```
b2FixtureDef fd;
fd.filter.categoryBits = 0x0002;
fd.filter.maskBits = 0x0001;
fd.density = 0.8;
fd.friction = 0.75;
fd.restitution = 0.1;
```

3.3.1 Complexity of Physics

In order to achieve good looking physical behavior, different parameters needed to be tuned, like coordinates, angles, and physical properties like density, friction and restitution. Specifying all these parameters at compile time made it compulsory to test the program frequently to see if the simulation performed as intended. This resulted in an increased development time.

Algorithm 3.2 shows how properties for the character's left leg are set. The same procedure is needed for every part of both rag dolls, for every part of the ropes, and for the ball. All these parameters affect the simulation in unpredictable ways. For example, assigning a low density value to some parts of the rag doll may make it float in the air falling very slowly to the ground. Or if the ball's density is set too high, the characters hanging from the rope may move in all directions after the ball hits them. This would make it hard to throw the next ball in the correct direction. The effects of the different parameters needed to be tested visually.

Defining coordinates can be done accurately from the source code, but this is only a good idea when the coordinates are known beforehand. The ball for example, should be exactly in the bottom-middle of the screen, so its position could be easily specified from the code. Cave polygons, for example, had to be positioned consistently with the background image. While this could have been done with little effort visually, positioning them from the code implied the tedious work of changing coordinates and testing if their position looked correct and did not overlap with other elements like the balcony.

A common problem with physics simulations happens when placing heavy objects on top of light objects, or when making heavy objects hang from lighter objects. The problem is known as *instability*, and it causes the simulation to produce random results. Because the rope was modeled as many pieces chained together, the second stability problem made the rope break apart or move erratically. In this case, there was no way to predict at compile time, whether the simulation would be stable. The simulation had to be run. So character's weight, joints between ropes, rope parts' physical properties, gravity and simulation number of iterations had to be tuned to find the optimal combination that delivered not only stable simulation values but that made physics behavior look realistic as well. Simulation stability can be improved by increasing the number of iterations the physics engine performs to update positions and velocities. But this affects performance. Therefore, not only correct physical behavior has to be tested, but also efficiency.

Algorithm 3.3 Left leg joint definition.

```
b2RevoluteJointDef jointDef;
b2Vec2 anchor(xpos/PTM_RATIO, ypos/PTM_RATIO);
anchor.y += leftLeg.sprite.contentSize.height/2/PTM_RATIO;
jointDef.Initialize(torso.body, leftLeg.body, anchor);
jointDef.lowerAngle = -0.4f * b2_pi;
jointDef.upperAngle = 0.0f * b2_pi;
jointDef.enableLimit = true;
```

Joint definitions had the same problems; they needed angles and coordinates that where hard to set programmatically, and they caused random behavior when not set properly. Testing the character joints required not only running the application, but also making the character move to see its parts' reaction. Because this test was needed very often, extra functionality had to be implemented to make objects react to finger touches.

Every game element was modeled with simple squares or circles, with the exception of the caves. Approximating the caves with circles or squares would make the ball either enter the background wall instead of the cave, or splash on the empty space of the cave. An external tool called VertexHelper was used to define cave polygon coordinates. VertexHelper loads an image and lets the user graphically define a polygon directly on the image. Once the coordinates are, set the program dumps source code that can be used in the project. More about this tool is explained in the next Chapter. Code sample 3.4 shows the source code defining the collision polygon of one of the caves. Similar code is needed for the other four caves. A better approach would have been defining the coordinates in an external file, so recompiling would not be needed after redefining the coordinates. But the main problem is actually defining those coordinates. Not only coordinates needed to be set manually for each vertex of each cave, but the polygons had to be convex and their vertices should be ordered in a counter-clockwise fashion. Whenever any of these conditions was not met, the physics engine caused undefined behavior. It either crashed, worked normally, or in the worst case it worked normally and crashed unexpectedly after a while. This behavior made it hard to identify the origin of the problem. Furthermore, because the images were clipped out of a sketch of the game, the same work will have to be repeated with the final cave sprites. A tool to automatically calculate the collision polygon out of an image would have made it easier to create cave polygons.

3.3.2 Changes in the Requirements

Using the accelerometer for gravity is a feature that no stakeholder thought about during the first meeting. It was, however, suggested by some of them and by other testers as well after seeing a prototypical implementation of the Rope game. This feature was not implemented yet, but will introduce problems with other dynamic objects, like the ball. The ball should not react to gravity before the player throws it, otherwise it would not remain in the center of the screen. Even if a solution to this

Algorithm 3.4 Collision polygon code generated by VertexHelper for a cave.

```
vertices[0] = b2Vec2(-3.5f / PTM_RATIO, 79.0f / PTM_RATIO);
vertices[1] = b2Vec2(-16.6f / PTM_RATIO, 71.6f / PTM_RATIO);
vertices[2] = b2Vec2(-31.5f / PTM_RATIO, -1.9f / PTM_RATIO);
vertices[3] = b2Vec2(-30.8f / PTM_RATIO, -78.7f / PTM_RATIO);
vertices[4] = b2Vec2(14.1f / PTM_RATIO, -78.7f / PTM_RATIO);
vertices[5] = b2Vec2(30.4f / PTM_RATIO, -54.3f / PTM_RATIO);
vertices[6] = b2Vec2(17.3f / PTM_RATIO, 41.2f / PTM_RATIO);
```

problem can be found, integrating the changes at such a late point in time will result in overhead time.

Another example of a feature that remained unexplored until a later point in time was making the characters react to finger touches. This feature was initially introduced with debugging purposes. After presented to clients, these decided to keep it as a feature so its functionality had then to be adapted and integrated.

The original sketch of the game had a fifth goblin in the top center of the screen. This can be seen in figure 3.1 in the Introduction chapter. For some reason the clients decided to remove that goblin and its balcony, probably because they thought it would be hidden by the hanging rag dolls. They communicated this to the development team after approximately one month of development, when the rag dolls had already been positioned. Because the missing goblin generated an empty space in the middle of the scene, the rag dolls and the ropes had to be repositioned. Doing this implied some overhead time, specially because it had to be done programmatically instead of visually. Later, during the final stages of the development and after the clients experienced a more evolved version of the game, they decided to add the goblin back and to push the dolls and the ropes far from each other again. A prototype would have helped stakeholders agree upon the game layout in the first meeting.

The created prototype together with usability tests revealed that the playability of the game needed to be improved. Players could not easily grab the ball, and once they grabbed it, they could not control it. There is a control area where the ball reacts to finger touches. In this area, the finger should be moved with the appropriate speed towards the bag. Children that tested the game tried either moving the ball too fast or too slow. To fix this usability issue the ball's physical properties need to be modified, or it should be treated outside the simulation. While the first solution implies once more testing the runtime behavior of different physical parameters, the second implies much more effort in making objects outside the simulation interact with simulation objects.

Chapter 4

Requirements Specification

Based on background theory and on the case study, this Chapter will identify and describe the requirements for the proposed system.

First, the Related Work section describes different tools and techniques related to quick prototyping, (serious) game creation, and physics editing. The objectives for the proposed system are directly derived from observations and ideas made in the Related Work Section. The other sections in this chapter describe scenarios and identify functional and non-functional requirements based on the system's objectives. Finally, an initial analysis model is described and a sketch of the user interface is presented.

4.1 Related Work

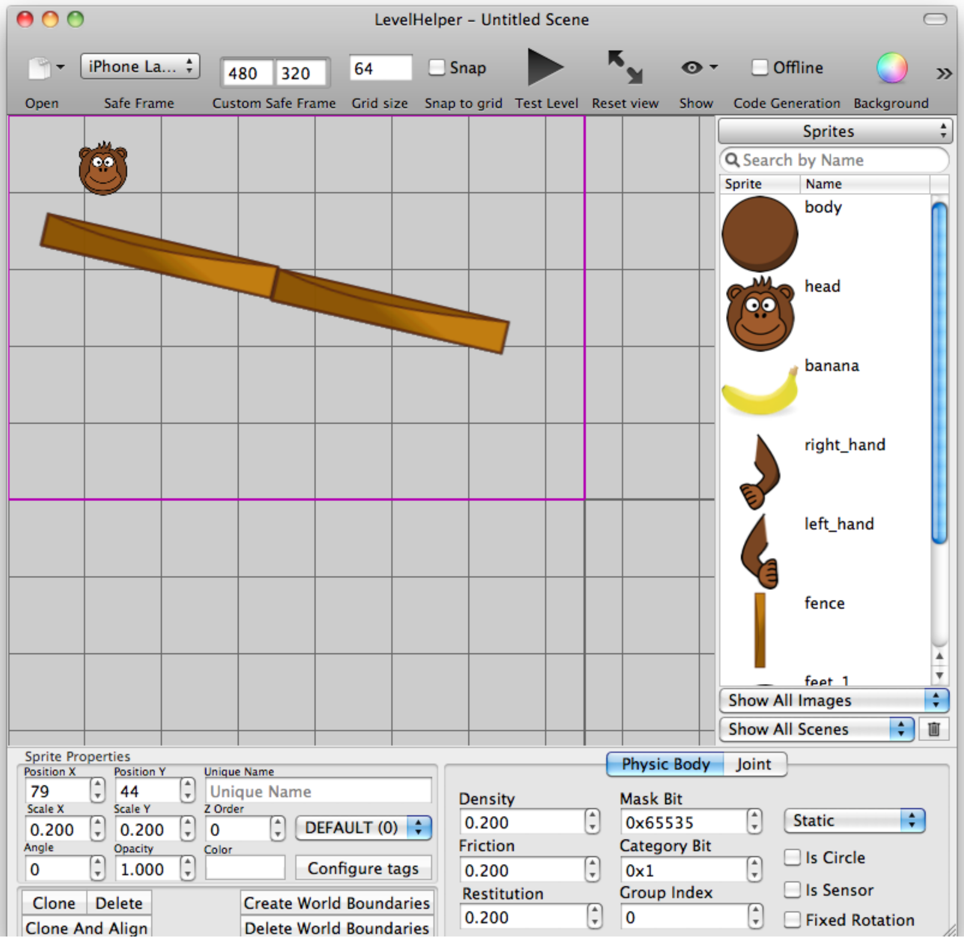
4.1.1 Physics Editors

A physics editor is a tool that allows developers to visually and interactively define the physical objects in an application. Figure 4.1 shows the GUI of a popular physics editor called LevelHelper.

Basic functionality physics editors offer consists on *transforming* objects (positioning, rotating or scaling) and *editing* object's properties (mass, friction, restitution, etc.). More complex functionality is often present, like letting the user define flags that determine which objects should collide with which other objects, or defining joints. Joints are used to constrain bodies' physical behavior in a certain way. For example, two objects may be joined such that they rotate around a common point, called anchor or hinge point.

The physical properties defined via physics editor are exported either directly to source code that can be integrated into developers' own projects, or to an intermediate format like XML that can be parsed and loaded from the code. Algorithm 4.1 shows how behavior is added to an object defined in a physics editor called "LevelHelper". The first step is to identify the object. This is done in line 4 with the label "Player-Name". The label has to match the "Unique Name" label provided in the GUI (see Figure 4.1 at the bottom left part of the GUI).

Physics editors are based on the fact that a lot of time can be saved in the creation of an application with real time physics, by using a WYSIWYG editor rather than



iPhone La... 480 320
Grid size: 64
Snap: ☐
Offline: ☐

Sprites: body, head, banana, right_hand, left_hand, fence, feet_1

Position X: 79
Position Y: 44
Unique Name: Unique Name
Scale X: 0.200
Scale Y: 0.200
Z Order: 0
DEFAULT (0)
Angle: 0
Opacity: 1.000
Color:

Density: 0.200
Friction: 0.200
Restitution: 0.200
Mask Bit: 0x65535
Category Bit: 0x1
Group Index: 0
Static
☐ Is Circle
☐ Is Sensor
☐ Fixed Rotation

Figure 4.1: GUI of the LevelHelper physics editor.

Algorithm 4.1 LevelHelper usage.

```
lh = [[LevelHelperLoader alloc] initWithContentOfFile:
                                    @"FileName"];
[lh addObjectsToWorld:world cocos2dLayer:self];
b2Body* player = [lh bodyWithUniqueName:@"PlayerName"];
//add behaviour to player
```

coding it.

There exist already a few physics editors. The most popular ones are described next.

- **BoxCAD** http://www.brainblitz.org/BoxCAD/ is a very complete flash based tool, it supports different shapes, joints, collisions, etc. It also supports dumping source code.

- **LevelHelper** http://www.levelhelper.org/ probably the most complete physics editor with support for iOS applications. It is based on the Box2D physics engine and it supports all its functionality, including automatic polygon calculation, joints, collision flags, etc. It generates a few source code files that can be included into the project. It has also advanced features like creating bezier shapes.

- **GameSalad** http://gamesalad.com/products/features is a very complete game prototyping tool, with basic physics support. It lacks joints and polygonal objects. It integrates predefined behaviors like moving or rotating objects, changing object attributes, playing sounds or displaying text. Games created with GameSalad can be exported to different platforms, including iOS, and published to an online server with a few clicks.

- **Mekanimo** http://www.mekanimo.net/ is another physics editor, with the same physics functionality as BoxCAD, and more. It has advanced features like merging objects so they behave like they are a single object, constraining objects' movement to an axis, connecting objects to springs so they bounce, etc. Almost every action in the editor generates python code dynamically, that can be modified by the user at both, edition and simulation time.

- **Game-editor** http://game-editor.com/Main_Page is not really a physics editor but it is has interesting game creation features like path editing, behavior via scripting and by defining events and linking them to actions. It supports many platforms; Windows, PocketPC, GP2X, Linux, etc, but not iOS.

- **Sketch Nation** http://sketchnation.com/ is a commercial iPad application for the creation of games. Although the games follow a strict structure, it has been popular in the beginning because users could create their own games and publish them to the App Store without needing any programming or image editing knowledge. Part of the revenue of each created game goes to the creators of Sketch Nation.

- **iTorque 2D** http://www.garagegames.com/products/torque-2d/iphone is one of the most complex game engines for iOS. It includes many features, like multitouch and accelerometer support, networking for multiplayer games, particle effects, tile manipulation, performance profiling, etc. It has the most of the common physics functionality, and more advanced features have been developed by its big community. Games are first designed using a WYSIWYG editor and then scripts can be used to edit objects' properties and behavior, while the tool takes care of the rendering.

- **Unity** http://unity3d.com/: "This is a full-featured 3D game engine with support for a wide array of features, including physics, terrain, and audio. It also includes a powerful world editor for set- ting up your scenes. Used for desktop and console games for years, Unity was recently ported to the iPhone" [3].

4.1.2 Sprite Helpers

Sprite helpers are tools used for the development of applications with real time physics to calculate the collision polygon of a sprite. If a sprite has transparent pixels, its collision polygon is the polygon surrounding all its non-transparent pixels. Collision polygons are needed by physics engines to detect and compute collisions.

- **Physics-Editor.de** http://www.physicseditor.de/ is a tool that targets the Windows and Mac platforms. Although its name suggests editing physics, its physics edition functionality is limited to editing objects' properties and collision flags. Its main feature is the automatically calculation of the collision polygon of a sprite, letting users select the amount of vertices of the resulting polygon. This is illustrated by Figure 4.2. The tool comes with a library, used from the code to load in a few lines the sprite and its physical properties.

- **SpriteHelper** http://www.spritehelper.org/ is a MacOS application. It is similar to Physics-Editor.de in that it calculates collision polygons, and lets users define a few physical properties. Additionally, it lets users pack textures into a bigger file that can be loaded at runtime faster than many small files. The tool automatically reorders and rotates sprites in order to reduce the size of the final image. The file containing all the sprites can be loaded from the code.

- **Box2d-editor** http://code.google.com/p/box2d-editor/ is a java application to automatically calculate collision polygons. The tool distinguishes itself from the others in that it includes a testing environment similar to those present in physics editors. The testing environment lets users throw balls against the previously generated collision polygon in order to test collisions. It uses Box2d for the physical simulation, but it exports the polygons into an engine-independent format.

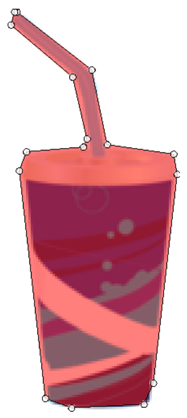


Figure 4.2: Editing the collision polygon of an image in Physics-Editor.

4.1.3 Braille Games Generator

[37] propose a semi automatic game generator for visually impaired children. The games it creates are very basic 2d-grid-like games, where different cells have different states. The states can be blocked, free, occupied by the player, occupied by an enemy, etc. Players and enemies can move through the grid.

Game developers first define how the grid looks like, the game states, and the way the player is allowed to move in the grid. It supports two way of movements, the *direct* way where the player moves up with the up arrow key, down with the down arrow key, and so on. And the *rotational* way, is the one used in the “Snake game”, where up and down arrow keys are used to move forward and backwards respectively, and side keys rotate clockwise and anti clockwise the direction of the player. Because the Braille terminal has a limited amount of cells, a window around the cell occupied by the player needs to be defined. The cells in the window are then “displayed” simultaneously by the Braille device. This window can be defined in the tool. Finally, the tool supports some basic behaviors that take place when the player enters a cell, like finishing the game and increasing or decreasing the score.

Once the game structure and some basic behavior has been defined using the tool, a code skeleton in python is generated. From this skeleton, developers need to place the elements in the grid (which is not yet supported by the tool itself), add additional behavior to the player object and create a Braille representation. A Braille representation defines how the cells in the visualization window map to Braille cells.

4.1.4 Parallel Software Generator

DPnDP [38] is a pattern driven tool for the creation of parallel applications. It offers a GUI from where the user defines the structure of the system by using a set of predefined design patterns or reusable modules. The structure takes the form of a directed graph, where every node of the graph can either be a design pattern or a module. Design patterns are composed by other nodes in order to form more complex communication structures. Modules represent sequential behavior and have the same interface of a design pattern. Any module can be refined (known as *hierarchical* *refinement*) into a more complex structure without affecting the rest of the system. This is illustrated by Figure 4.3, where one of the nodes is internally refined into a more complex structure.

Once the graph has been created, the platform generates a code skeleton in C++ that takes handles the entire communication and synchronization needed by the application. To the generated code, the application dependent behavior has to be added.

One of the most important properties of the system is extensibility. DPnDP was the first platform to handle patterns in an abstract way (known as *generic view*). This allows new patterns to be easily added to the system, as soon as they are compliant with the pattern interface.

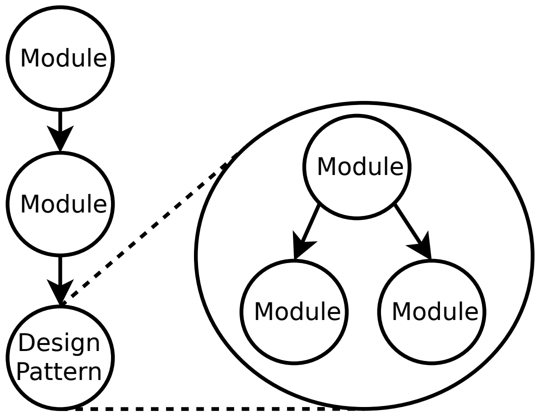


Figure 4.3: DPnDP communication model [38].

4.1.5 Software Cinema

Brügge et al [5] propose a different way of improving requirements elicitation, based on movies. In Software Cinema, movies have the same purpose as those identified by [11] for prototypes: to provide a better understanding of how an envisioned system may *look*, *work*, and be *used*.

The technique consists on defining spatial and temporal relationships between different elements of the video. Elements in the video can be identified and made “clickable”. Descriptions are then added and linked to the elements. When the audience desires more details on a specific object, they can click on it, and navigate to other elements. Annotating a film gives place to a fine-grained level of detail, which does not come at the price of an abstract representation that only engineers can understand.

Objects, their descriptions and relations to other objects belong to a *conceptual model*, which can be understood by anyone. The Software Cinema Toolkit is able to map the conceptual model to a *formal model*, which is used by engineers to define the system's specification. Each scene in the conceptual model can be mapped to a UML use case diagram, and sequences of annotations can be used to generate sequence diagrams. The Software Cinema technique tries to fill the gap between end users and engineers by allowing each stakeholder to use the model that best suits its background knowledge and its role in the project.

4.1.6 Scratch

Scratch [33] is not only a web-based tool but also a social network for creating and sharing games, animations and any kind of interactive media. Its target users are children with little to no experience in programming. It supports graphics, animations, photos, music and sound objects.

It is based on programming blocks. A block can represent variables, methods, flow control statements like if or for loops, etc. Each different kind of block is represented by a different color. The blocks also have different shapes and can only be inserted into matching shapes, like in a puzzle. For example, for loops have a C shape to suggest that something goes inside. Number variables are ovals, so methods that receive numbers as parameters have an oval-shaped slot. Hexagons represent booleans, so conditional blocks have hexagonal slots. Figure 4.4 shows an example of different blocks puzzled together.

Scratch is highly interactive, for example clicking on a block executes its code in the integrated simulation environment.

This tool mixes high level objects like images and animations that are ready to use, with low level programming which is also supported by blocks with higher level semantics.

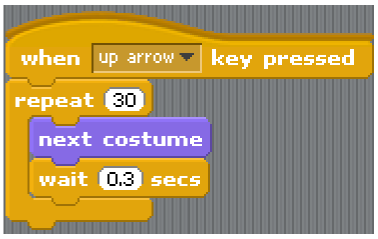


Figure 4.4: Blocks in Scratch.

4.1.7 Lua

Lua is a dynamic typed scripting language. It was designed to be simple, small, portable and easy to embed into applications [14]. It is a library written in ANSI C with 10000 lines of code. It does not have the concept of a main routine, it is rather meant to be embedded in a host application. Its reflexion mechanisms make the integration with other applications easy and allow new code to be added dynamically to the application.

It is a first-class language, which means that any object and any subroutine in a program can be constructed dynamically, stored in variables, passed as parameters to subroutines, or used as return type of subroutines.

In Lua, data structures are defined using the so called *tables*. Tables are associative arrays; they are composed of keys used as index, and elements. Tables can be used to define many different kind of data structures, like sets, generic linked structures, etc. There are different ways of constructing tables. One example is the { } operator, as illustrated by Algorithm 4.2.

Lua has a semantic extension mechanism called *fallbacks* [7]. Fallbacks are used to set the control flow of an application when Lua does not know how to proceed (comparable to try-catches). Some situations when this may happen are when applying arithmetic operations between non-number objects or when an order comparison is applied to non-numerical or non-string operands [12].

Algorithm 4.2 Defining a table in Lua.

```
{lat= −22.90, long= −43.23, city= "Rio de Janeiro"}
```

Lua has great utility in the creation of prototypes programmatically.

4.1.8 Summary

With a very few exceptions, the different game creation tools analyzed in the previous section run on desktop computers. In order to test the created application on the target device, an intermediate step is needed. Current tools accomplish this in two different ways. One of them is by generating source code that can be manually integrated into a running application. The advantage of this is that more complex functionality can be added by programming. The disadvantage is that time is lost integrating the editor-generated code with self-written code. The other way they let users test the designed application is by simulating the application on an integrated platform simulator / emulator. Thanks to this, an application can be quickly tested after being created / edited. However, this limits the usage of device specific capabilities, such as multitouch and accelerometer input. SketchNation runs directly on the iPad, but it is not a general purpose prototyping tool. Its functionality is very limited to a very specific type of games.

Most of the current game creators, and particularly those that only serve the purpose of physics editors, are not able to create quick prototypes. On one hand, it takes some time before an application can be tested on the final device, as already described. On the other, they offer advanced features (polygonal and bezier shapes, profiling, different kinds of joints, etc.). This is provided at the cost of a crowded GUI, with functionality that is sometimes unclear even to expert users and ends up never being used. There is also no tool with support for iOS that lets users define behavior (other than via source code), which could be more useful for an initial prototype, rather than the creation of complex structures. GameSalad is a quick prototyping tool and has functionality for defining behavior, but it only creates throwaway prototypes.

Every tool with physics support generates plain structures. This makes the task of adding behavior to concrete objects hard. LevelHelper for example lets developers label single objects. Labeled objects can then be identified with the same label in the source code. This makes it easier to add behavior to these objects. But it also leads to non-maintainable code because of hardcoded strings in the code that have to match labels in the editor. Whenever an object is removed or a label changed in the editor, developers need to find the code where the object is loaded from the external file and adapt it accordingly. DPnDP realizes this issue and uses a hierarchical structure.

This makes created applications more extensible and structures defined by the user reusable.

Physics Editors and Sprite Helpers functionality is separated from each other. There is no tool that integrates both. LevelHelper gets the closest to the need of a multi-purpose tool. It accepts packed textures generated by SpriteHelper that can be placed in the scene and treated as physical objects. But still two different tools have to be used and users have to deal with the output files generated by each of the tools. Ideally, a tool would let the user select sprites, calculate collision polygons, and edit physical properties all at the same time.

Software Cinema tries to bridge the gap between application and solution domains. A conceptual model that anybody can understand (in particular, end users) is automatically mapped to a formal model needed by developers.

4.2 Objectives

The proposed system should improve existing systems in the following ways:

1. It should run directly on a tablet. This presents several advantages.
   (a) Created applications can be tested without overhead time writing code and making use of every device capability, with a real user experience.
   (b) A tablet is more portable than a laptop or desktop computer, so it can be taken to every meeting with clients and end users.
   (c) It also offers more direct ways of interaction, like touching the screen. This makes an application running on a tablet more likely to be accepted by clients with no IT experience.
   (d) Tablets promote collaboration of different users. As opposed to laptops and desktop computers, they are meant to be used by more than one person at the same time.
2. The purpose of the system is to create prototypes quickly. The prototypes do not need to be complete or accurate. The system should have basic functionality for defining structure and also behavior, rather than focusing on structure. More accurate and advanced features can be added to the prototype later in the code.
3. Created prototypes should belong to the evolutionary category in order to benefit software projects from the advantages of prototyping, while avoiding throwing away work done.
4. Created structures should be hierarchical in order to favor reusability of components and extensibility of created applications. This suggests an additional advantage when compared to current physics editors, in that behavior can not only be added to individual objects, but also to complex structures.
5. The system should integrate functionality of physics editor and sprite helper. It should be possible to edit the physics of an application and to calculate collision polygons at the same time.

6. In the same way as Software Cinema is able to map a conceptual model to a formal model, the proposed system should be able to transform application domain semantics into solution domain semantics. The idea is the same as the one behind [25], although it is achieved in a different way. Instead of creating a single language that merges advantages of prototype and class based languages, the proposed methodology includes both languages as they are, but uses them in the phases of the software engineering lifecycle where they are best suited. This is illustrated in Figure 4.5. PBLs are closer to the application domain and are therefore best suited for requirements elicitation. During this phase, any user should be able to interactively define clonable prototypes out of some basic elements such as squares, circles and images. The system should then map the model defined using prototypes to a formal model based on classes, that is best suited for development. This is illustrated in Figure 4.6.

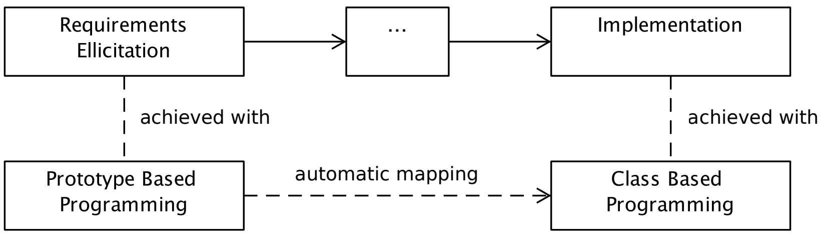


Figure 4.5: Usage of prototype and class based languages in different phases of the software engineering process.

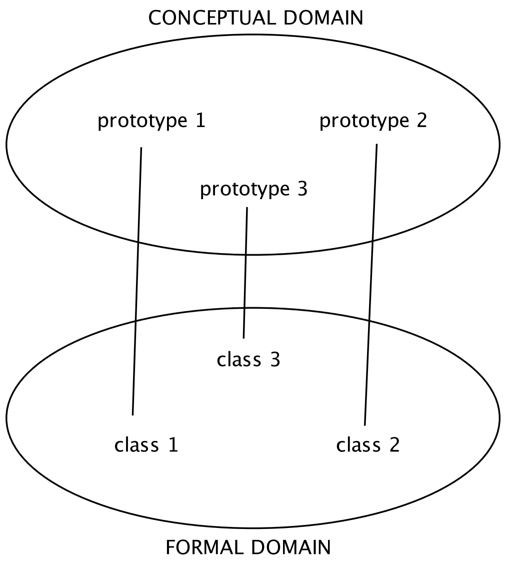


Figure 4.6: Mapping between a conceptual model and a formal model.

4.3 Scenarios

In this section, the first and second problem scenarios will reinforce the problems described in (Case Study 3). Afterwards a visionary scenario is described. All three scenarios correspond to the same software problem, where Abel, a game developer, uses different technologies to provide a solution.

4.3.1 Problem Scenarios

Scenario A - 5 months of work

Abel meets his client to talk about requirements for a new game. The client uses some scribbles to explain what the game is about and how it is played. The game is called rag-dowling; it is a bowling game where instead of bowls there are rag dolls. There are three rag dolls and the player uses his finger to throw the ball against the dolls. The player scores as many points as dolls are not standing after the ball is thrown.

After the meeting, Abel goes home and starts thinking about the game. He realizes he will need to use a physics engine. He spends three months implementing the game. Great part of the time is spent programmatically defining collision polygons and setting positions of objects and joints.

When he thinks the game is ready he organizes a new meeting with his client, only to find out that some requirements had been misunderstood and that the users of his game have problems to interact with it. As a consequence Abel will need two extra months of work.

Problem:

- Abel has invested great part of his development time configuring the structure of the game. He had the scribble his client gave him, and he tried to copy it programmatically, which is a tedious work. Whenever he did a change in the code, he was not sure whether it would look good so he needed to run the application and tune the parameters many times.

- The waterfall model assumes that all requirements can be elicited in the first developer-client meeting, which is almost never the case. Misunderstandings between experts in the solution domain, and experts in the application domain are the rule and not the exception. Additionally, it is extremely hard to predict how good users will be able to interact with the software. Both problematics cause change in the requirements. As we know from [2], *change* is one of the major problems in software engineering, specially when it happens together with *complexity*. Unfortunately, complexity happens inevitably as a project evolves. The problematic could be solved by making change happen in the very beginning of the software project. In this scenario, requirements change three months after the project started, causing a big overhead.

Scenario B - 4 months of work

The same scenario as above, but this time Abel decides to use LevelHelper since it is the most complete and popular physics editor in the market.

After the meeting with his client, Abel goes home and starts creating a rag doll using 20 different polygons. In order to identify the doll from the code, he labels all 20 objects in the editor and then goes to XCode and writes 20 lines of code to load the doll from the external file. Although the other dolls are identical, he cannot simply instantiate two more rag dolls. He needs to repeat the previous work to load all 20 parts of each rag doll with their corresponding names. He is able to finish the project within two months. As in the previous scenario, some requirements had been misunderstood from the beginning and there are some usability problems to be fixed, causing two extra months of work.

Problem:

- Current physics editors promote the waterfall development process. After the meeting, Abel needs to go home and come back later with a prototype. While using a tool to create the prototype improves the previous scenario, it could be even better if the prototype would be created and tested directly and interactively with the client (and / or end users).
- Existing physics editors generate *plain structures*. This means that games are composed by a set of independent elements instead of a hierarchy of them. This affects reusability. Abel has to create three rag dolls part by part instead of creating a rag doll prototype that can be cloned three times. It is also impossible to add behavior to the rag doll itself. Instead, behavior has to be linked to each part individually.
- The approach most tools enforce to add behavior to each element is tedious and error prone. Elements are labeled in the editor. The same label is then used from the code to identify the element. So labels in the editor and code should be kept consistent. If an element is removed or its label changed, the code should be updated accordingly. If a wrong label is coded, the application crashes. Reusability is also affected. If two extra rag dolls are needed and they should have the same behavior as an already created rag doll, it is not enough to copy all their parts in the editor. All their labels have to be renamed in the editor and the code adapted accordingly.

4.3.2 Visionary Scenario

Scenario C - 2 months of work

The same scenario as above. This time the market offers a new tool that runs directly on an iPad. Abel has it and takes it to the first meeting with his client. While the client is explaining the game, Abel pulls out his iPad and starts creating a raw version of the game, as he understands it. As the client explains, Abel interactively adds elements to the game. By observing Abel, the client is able to understand the simple logics behind the tool. So when he is not satisfied with how some elements look or when he is not able to explain what he wants with words, he directly grabs the device and finds a way to graphically explain Abel what he has in mind.

Thanks to the prototype that lets them see an overview of the game, they get new ideas that should increase the quality of the game. The prototype also has features they did not expect. The client likes some of these features and decides to include them in the final game. He also indicates what other features should be different.

When the meeting is over, Abel goes home, continues refining the prototype, creates a rag doll prototype and then uses the tool to generate a source code skeleton. The generated code is object oriented, so the rag doll prototype maps to a `RagDoll` class, where Abel implements the corresponding behavior. Because all three rag dolls in the game are instances of the RagDoll class, behavior is inherently in each of the instances. Thanks to this Abel does not need to explicitly link structure in the editor with behavior in the code nor to manually keep those links up to date.

Development time is now of only one month. When Abel meets the client again, only a few details have to be improved, for which an extra month is needed.

4.4 Functional Requirements

This section describes the functionality of the system.

4.4.1 Editing

1. The tool should let users build the structure of the game. For this, users drag with their fingertips objects from a palette into the scene. The following objects should be available:

    (a) (Non-) physical object. Non-physical objects are images that do not participate of the simulation. Physical objects have a shape (square, circle, line or polygon) and react physically to forces and collisions with other objects. Properties of physical objects are density, friction and restitution. They also count with collision flags used to determine with which other object they should collide.

    (b) Joint. Joints will be used to make two objects rotate around a common point. Properties of the joint is the minimum and maximum rotation angle.

    (c) Animation. Animation objects are collections of images that get linked to one or more (non-) physical object(s). When started, an animation exchanges the object's image with each of the images in the collection, at a given speed. Animations have a looping count (how many times the sequence of images is repeated) and the animation speed. Animations can also restore objects' original image.

    (d) Sound effects and background music objects.

    (e) Particle system. A particle system used for game effects such as explosions, fire, etc. Particles are emitted at a (random) *position* and with a (random) *velocity* and start *vanishing* and *decreasing* in size immediately after being created. Particles die when their size is close to zero, when their life is smaller than zero, or when the particle system has been active *duration* or more time. The particle system should contain the following editable properties:

i. Number of particles.
ii. Emission rate. Number of particles emitted per second.
iii. Duration. Every particle should die after the duration.
iv. Initial and final color. As the particles vanish, their color and transparency levels should be interpolated between initial and final colors.
v. Initial life and initial size of the particles.
vi. Vanishing speed (how fast they die) and diminishing speed (how fast their size decrease).
vii. Initial position where they are emitted (in pixels) and a position variance (maximum number of pixels by which the initial position of each particle can differ from the user defined initial position).
viii. Initial velocity and initial velocity variance (maximum number of pixels by which the initial velocity of each particle can differ from the user defined initial velocity).
ix. Potential function. When set to *none* particles will not interact with each other. Any other value will cause particles to interact with each other according to distance between each other and the user selected potential function.

(f) Action. Actions are predefined behaviors such as making objects move or changing object's sprites. The system should support the following actions:

i. Move action. Moves an object either by a relative distance or to an absolute position.
ii. Scale action. Scales an object.
iii. Create action. Makes an actor appear in the scene.
iv. Destroy action. Makes an actor disappear from the scene.
v. Sprite action. Changes an object's sprite.

(g) Trigger. Triggers are what cause an action to be executed. They can also be understood as rules: "Whenever this trigger's condition is fulfilled, these actions execute". Triggers are connected to one or more actions. Triggers can also be connected to animations or sounds, to make them start playing. The user should be able to choose whether a trigger should execute the first time the condition is met, or always. The following triggers should be supported:

i. Time trigger. A timer that triggers actions when it reaches zero.
ii. Position trigger. It triggers whenever one or more objects reach a specified area.

iii. Scale trigger. It triggers whenever one or more objects reach a specified size.

iv. Collision trigger. It triggers whenever one or more objects collide against other objects.

2. Object transformations should be done with the multi-touch capabilities of the device. Objects are moved with the *pan* gesture, rotated with the *rotation* gesture, and scaled with the *pinch* gesture (see: 2.3). Objects are selected for editing with the tap gesture.

3. Every object has different properties, for example a square physical object has the width and height properties. The particle system object has the number of particles, the emission rate, the color of the particles, etc. The system should allow users to edit objects' properties.

4. The system should let the user group basic objects (squares, circles) into composite objects. Grouped objects are treated as a single object. They can be transformed and they have their own editable properties. A composite object must be ungrouped before its parts' properties can be edited.

5. The system should let the user categorize objects by giving them a name. The category is just another editable property of every (non-) physical object.

6. Categorized objects can be dragged into the palette. Once dragged in the palette, they act as the other palette items; they are prototypes that can be dragged into the scene in order to be cloned.

7. The system should allow created applications to be saved persistently.

8. The system should be able to compute a collision polygon out of an image.

4.4.2 Simulating

1. The system should let the user test the application in any moment.

2. Because during simulation time object properties may be altered, the system should save every object's state before the simulation.

3. The simulation should look exactly as the final application, without any extra element, not even a button for ending the simulation. The transition between simulation and the editing states should be done with gestures. The system should let the user select the gesture used to switch states. Possible gestures are: shake gesture, three fingers swipe, etc.

4.4.3 Generating Code

1. The system should generate the necessary source code to make the application run exactly (without taking performance into account) as while being simulated.

2. The generated source code should be object oriented. When code is generated, palette items should map to classes, and objects should map to instances of those classes.

4.5 Non-functional Requirements

This section describes the qualitative aspects of the system that receive the name of non-functional requirements [2]. Usability, performance, supportability, robustness and implementation requirements are identified.

4.5.1 Usability

1. It should be possible to perform the following actions with at most one multi-touch gesture.
   (a) Adding objects to the scene.
   (b) Rotating and scaling an object.
   (c) Joining two objects.
   (d) Starting the simulation.
2. It should be possible to perform the following actions with at most two multi-touch gestures.
   (a) Removing objects from the scene.
   (b) Editing an object's properties.
3. If the palette items are not big enough, it may be hard for the user to tap them with the finger. The palette items should have a tappable area of at least 50x50 pixels.
4. Before removing objects the system should ask the user for confirmation. This should prevent cases where the user accidentally tapped the remove button when trying to edit object's properties.
5. In order to minimize the chances that users tap the remove button when trying to tap somewhere else, the remove button's tappable area should be 50 pixels away from any other widget's area.
6. Sounds in the palette should not only display a label to let the user identify them, but it should also be possible to play them with a single tap.
7. Icons should be chosen according to the iOS usability guidelines. Apple offers ready-to-use icons that should be used consistently across applications in order not to confuse users.
8. Generated source code skeletons should compile and run without further changes.

4.5.2 Performance

1. The system should provide immediate feedback to the user when object properties are being edited, or when the object is being transformed. This requirement will be fulfilled if performing any editing task does not make the frames per second drop below 30.

2. The integrated simulation environment will cause simulated applications not to perform as good as they could when run independently. One reason is that the application will have less memory available when running from the simulation environment. This requirement will be fulfilled when the simulated application has a performance of at least 95% its performance when running independently. Performance should be measured in terms of average FPS over the time it takes to use every feature of the created application.

3. Switching between editing and simulating states and vice-versa should be fast so the user can test as often as he wants. Switching from editing to simulating states may be particularly time consuming. This is because the state of every object in the editor needs to be saved, and physical objects loaded (in case the object used for simulation differs from the object used for editing). This requirement will be fulfilled if switching between states requires less than 0.3 seconds with up to 50 objects in the scene. If, in order to fulfill performance requirement 2, objects in the editor should be saved in persistent memory, they could be saved immediately after being created or edited instead of all together before simulation. This would prevent the application from freezing when switching between editor and simulator. However, special care needs to be taken in order not to violate performance requirement 1. For example it would probably violate it updating persistent data in response to continuous events such as the ones generated by sliders, or the ones produced by multi touch gestures.

4.5.3 Supportability

Embedded device technologies evolve particularly fast. Newer versions of frameworks, APIs and target operating systems are likely to be released even before the system reaches the market. A software design should make the system ready for new technologies and requirements with little effort.

While the scope of this thesis is an iPad application that supports the creation of other iPad applications that need physical simulations, it will most likely change in the future. The following requirements should reduce the amount of work needed when these changes occur:

1. The application domain may not have anything to do with physics. The system architecture should therefore isolate physics domain specific functionality from editing and simulating functionality common to any domain. For example, detecting a tap gesture and finding an object by touch location is common functionality to any domain. Rotating an object with rotation gestures is physics domain specific functionality. If, for instance, the tool is used in the future for

music creation, a music object may react to rotation gestures by changing its volume, rather than rotating.

2. The system may target other platforms than iPad. Boundary objects like palette views are likely to be replaced in the future and should be isolated from control and model objects.

3. The system may support the creation of applications for other platforms. Code generation mechanisms should therefore be extensible. This can be achieved by having different `CodeGenerator` subclasses and using the Template Method design pattern if all of the supported platforms' programming languages share a common structure, or with the Strategy pattern if they do not share a common structure. The code generation client should be isolated from the code generation method, which should be decided at runtime. Code templates should be used to avoid hardcoding source code.

4.5.4 Robustness

1. An application for creating other applications needs special care in order to prevent users from creating an application with abnormalities. This may be the case specially when using triggers. For example, if the user attaches a destroy action and a move action to the object, and the destroy action happens at simulation time before the move action, the system's behavior may become undefined, if the system was not prepared to cope with such a situation. Another example would be connecting a time trigger with a very high frequency, to a creation object. The application may freeze creating objects until memory exceeds the maximum allowed by the operating system and would then crash loosing non-persistent data. The system should prevent these issues. Defensive programming can help prevent some of them.

4.5.5 Implementation Requirements

1. The application should run on iOS and should therefore be implemented using the XCode IDE.

4.6 System Models

In this section the use case model and object models derived mainly from the functional requirements in section 4.4 are presented. The purpose of this section is to provide an overview of the system that will be refined in later chapters.

4.6.1 Use Case Model

Figure 4.7 illustrates what functionality the system should offer to its users, in the form of a use case diagram.

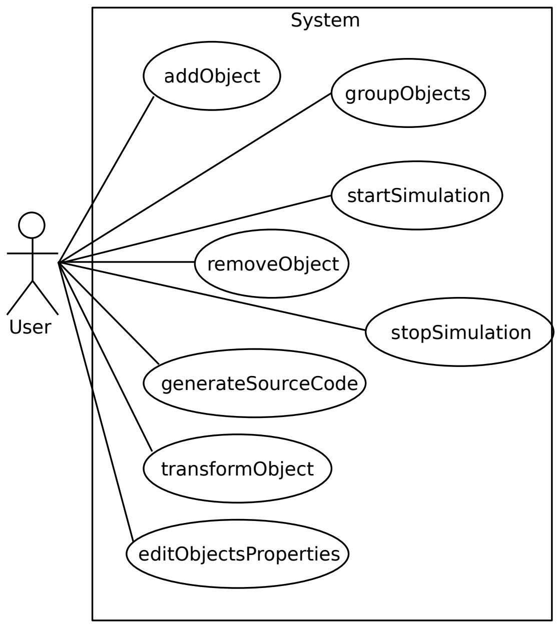


Figure 4.7: Use cases identified in a UML use case diagram.

4.6.2 Object Model

This subsection focuses on the internal structure of the system. Objects identified from the functional requirements and the relations between them are shown using class diagrams.

4.6.2.1 Editing and Simulating

The `Editor` class handles editing functionality such as applying transformations on objects, selecting objects, editing selected objects' properties, choosing simulation options such as gravity, etc. The `Simulator` class handles application's initialization and interaction between user and application. Because `Editor`, `Simulator` (and `CodeGenerator`) access and interact with the World, they inherit the `WorldAgent` class.

The `World` class is a model object that acts as container for application objects. It ensures the correct application's state when being edited by offering a well defined interface that encapsulates application state. This interface is used by the `Editor` for example to add objects, join them, etc. It also implements necessary behavior for the interaction between application objects at simulation time, which is used by the `Simulator`. Relations between `World`, `Editor` and `Simulator`, are shown in Figure 4.8.

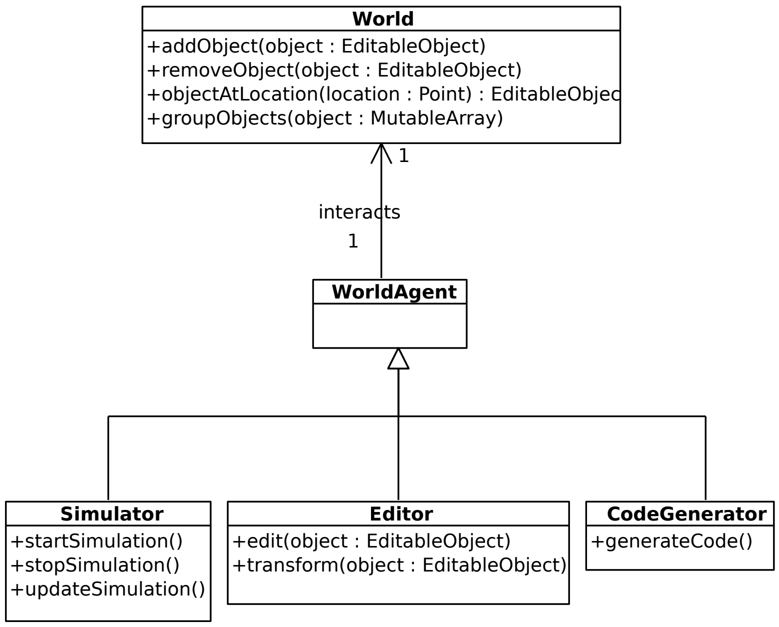

World
+addObject(object : EditableObject)
+removeObject(object : EditableObject)
+objectAtLocation(location : Point) : EditableObjec
+groupObjects(object : MutableArray)
1
interacts
1
WorldAgent
Simulator
+startSimulation()
+stopSimulation()
+updateSimulation()
Editor
+edit(object : EditableObject)
+transform(object : EditableObject)
CodeGenerator
+generateCode()

Figure 4.8: The `World` class and its relation to application objects, `Editor` and `Simulator`.

4.6.2.2 Palette

From the following requirement "*users drag with their fingertips objects from a palette into the application*" and using Abbott's method, the classes `Palette`, `PaletteObject` and `Scene` are identified. The `Editor` is composed by a `Palette` and a `Scene`, and a `Palette` is composed by many `PaletteObjects`. Figure 4.9 shows the `Palette` and the `PaletteItem` taxonomy. Note that there exists a `PaletteItem` subclass for every application object. The different `PaletteItem` objects instantiate different application objects.

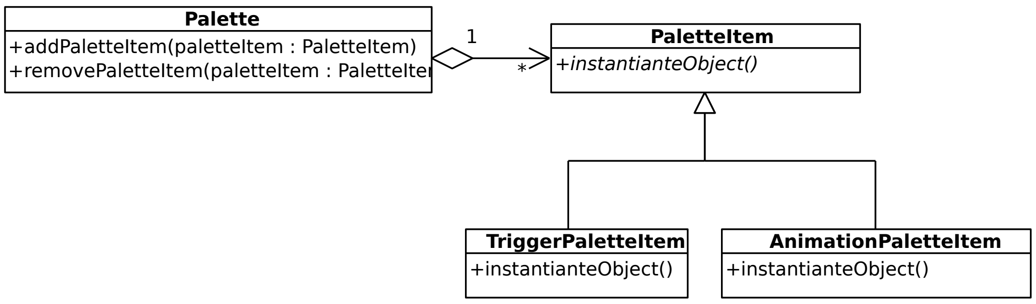


Figure 4.9: The `Palette` and its relation with the `PaletteItem` taxonomy.

4.6.2.3 Application Objects

Application objects are the components of the created applications: (non-) physical objects, joints, animation, etc. They share common functionality; they can be added and positioned into an application, and their properties can be edited. Therefore they all inherit from a common `EditableObject` class. Different `EditableObjects` override the `draw` method and the default transformation methods. This is shown in Figure 4.10. Note that not every application object is included in the diagram, for the sake of simplicity. Refer to Section 4.4 for a complete list of the application objects.

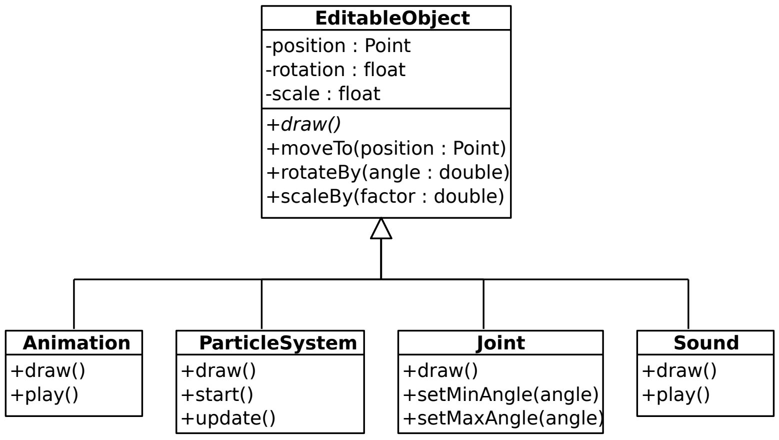


Figure 4.10: The `EditableObject` taxonomy.

The physical object is designed as a Composite Pattern, according to functional requirement 4 in 4.4.1 on page 34. This is shown by Figure 4.11.

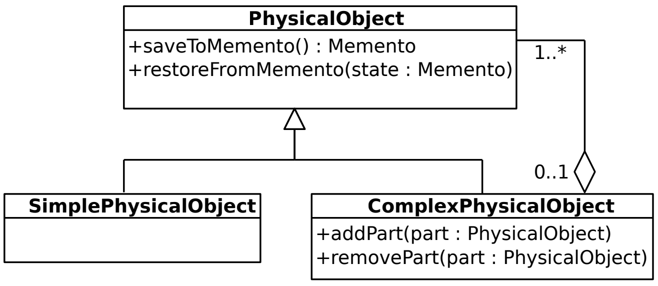

PhysicalObject
+saveToMemento() : Memento
+restoreFromMemento(state : Memento)
1..*
0..1
SimplePhysicalObject
ComplexPhysicalObject
+addPart(part : PhysicalObject)
+removePart(part : PhysicalObject)

Figure 4.11: The Composite PhysicalObject.

`Triggers` define two abstract methods, `shouldTrigger` and `trigger`. The first returns a boolean indicating whether the trigger's condition is met. When the condition is met, clients are allowed to invoke the `trigger` method. The `trigger` method triggers actions, sounds and animations. This is shown by Figure 4.12.

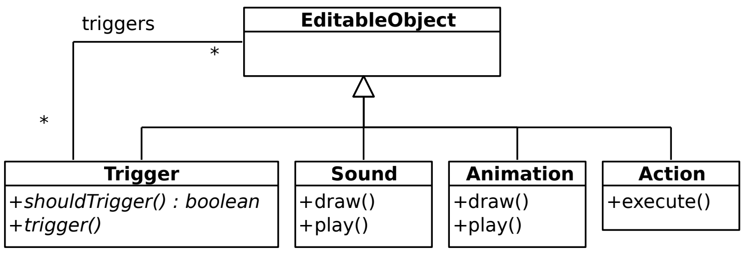


Figure 4.12: Relation between `Triggers` and other `EditableObjects`.

4.6.2.4 Code Generation

The `Template` class represents a code template file. It loads and parses the file, can replace some of its parameters by client specified strings (ex. the class' name), and can generate compilable source code. The `CodeGenerator` class uses one or more code templates to generate code. Each subclass of `CodeGenerator` generates source code for a different target platform, in order to fulfill supportability requirement 3 in 4.5.3 on page 38. The `CodeGenerator` taxonomy and its relation to the `Template` class is illustrated in Figure 4.13.

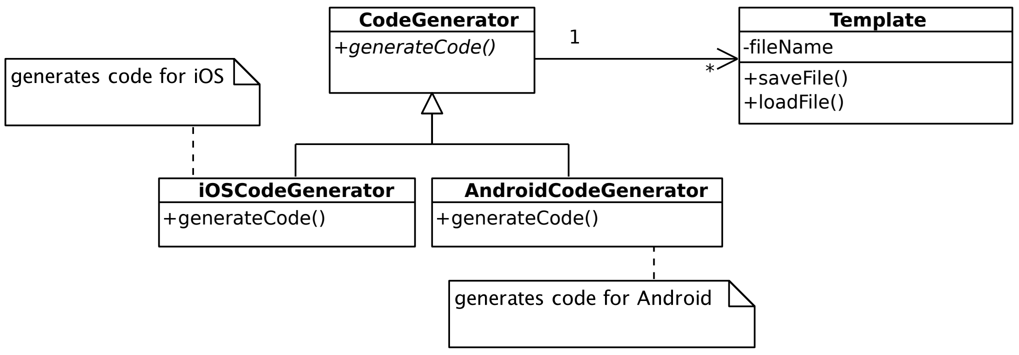


Figure 4.13: The `CodeGenerator` taxonomy and its relation to the `Template` class.

4.6.3 Dynamic Model

The object model covers the static structure of the system. In order to have a better overview of the system's preliminary design, this section describes part of the system's runtime behavior.

The system can be in three different states, as illustrated in Figure 4.14. Initially, the system is in editing state where the user can create an application by adding objects and modifying its properties. Once the application has been created, it can be directly tested on the device or it can be exported to source code. In order to test it, the user shakes the device. In this case, the system enters the simulating state.

To return to editing state, the device is shaken again. Source code can be exported from the editing state. Once the user provides the appropriate parameters, code is generated and a completion message is displayed on screen. Accepting the completion message leads the user back to the editing state.

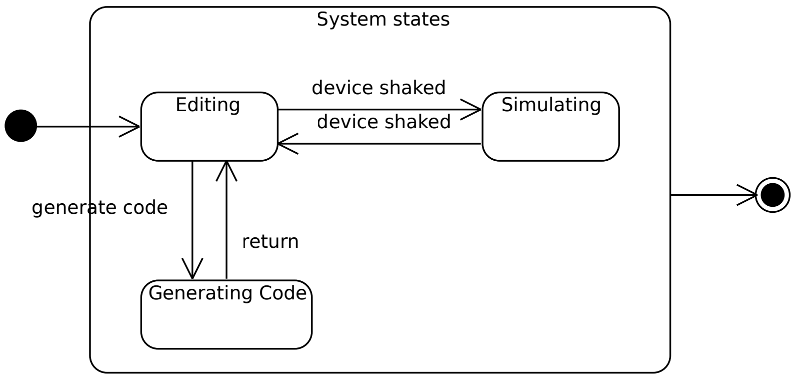


Figure 4.14: States diagram of the system.

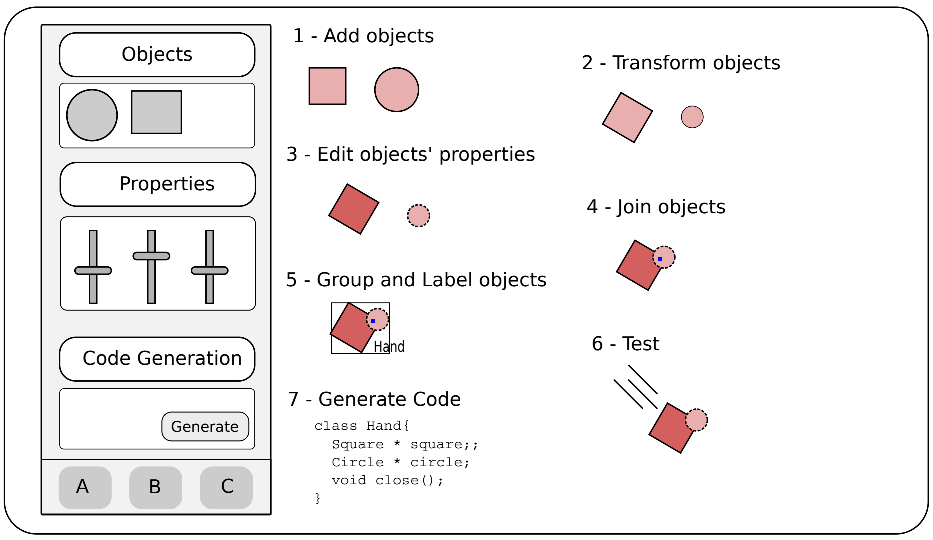
4.7 User Interface

This Section illustrates and describes the user interface for edition, which consists of the *application view* and a *palette.*

The palette has three tabs, A, B and C. Tab A corresponds to adding objects: physical objects, sounds, animations, etc. These objects can be dragged into the scene. Tab B is used for editing objects once added to the scene. Tab C corresponds to behavioral objects. Behavioral objects are triggers and actions and are used to provide behavior to the application. These objects can also be dragged into the scene. Instead of the labels A, B and C, the real interface should have images. The palette is semi transparent to let the user see what is behind it.

The application view is where every object added to the application is shown to the user. The user interface in simulation state has no additional elements, i.e. it looks exactly as the final application.

Figure 4.15 shows the user interface, together with a summary of the functional requirements.



```
class Hand{
  Square * square;;
  Circle * circle;
  void close();
}
```

Figure 4.15: Sketch of the system's interface, together with the functional requirements.

Chapter 5

System Design

This chapter shows the transformation of the analysis model described in 4.6 into a system design model. It presents the preliminary architecture along with the design goals that influenced it. The following design aspects are covered in this chapter.

- Design Goals (Section 5.1). The qualities of the system that should be optimized in the development.

- Enabling Technology (Section 5.2 on the next page). Existing tools, frameworks and components that can be used for the implementation.

- Subsystem Decomposition (Section 5.3 on page 49). The system's architecture and the scope of the different parts of the system.

- Persistent Data Management (Section 5.4 on page 50). The data that should be stored.

5.1 Design Goals

There is a potential conflict between some of the defined requirements which are considered in this section.

- Usability vs Functionality. A simple interface with few application objects and with few editable properties per object is likely to be more easily understood by users than a crowded GUI with advanced features. There already exist physics editors that implement complex functionality. One of this system's goal is to allow users to obtain results faster and easier than with the other tools, rather than offering more or better functionality. Priority should therefore be given to usability over functionality.

- Cost vs Robustness. Although robustness has been identified as a system requirement in (4.5.4) abnormalities of user generated applications will be accepted if the only way to detect them is to perform the simulation. This would require a non cost effective effort to monitor simulations' state, which should therefore be avoided.

5.2 Enabling Technology

This section presents a survey about technology that can be used for the realization of the system. These are mainly APIs to be used during the implementation phase of the project.

5.2.1 Cocoa Framework

The Cocoa framework is a group of APIs, runtimes and libraries that are tightly integrated into the XCode IDE. An application using the Cocoa framework immediately inherits behaviors and appearances of the Mac OS X. Some of its features, relevant for this system:

- Core Data is a framework part of Cocoa that can manage model objects in an application and its relations to other objects. Entity objects can be created visually, its properties and relations to other objects edited and the corresponding class can be generated with a few clicks. Any change done to a property of an entity object can be undone at any moment. Objects and all their relationships can be serialized with little effort.

- The Cocoa Framework comes with an integrated interface builder with GUI elements like tab bars, buttons and sliders needed for the palette do not need to be created from scratch. Another important reason to use these elements is the user familiarity with their appearance and behavior. The framework also minimizes the amount of code needed for the creation and integration of the GUIs. “Wiring” Controller logic which is coded with application Views is also done graphically.

5.2.2 Cocos2d Framework

Cocos2d is a framework for building 2D graphical/interactive applications. Because it is free and open source and because of its good trade off between high level functionality on top of OpenGL ES and performance, it is very popular among iOS game development community. Some of its features are:

Sprites	With only two lines of code a texture is rendered into the screen. The framework takes care of caching the texture once it was loaded from the persistency storage system. With another line of code the texture can be transformed: scaled, rotated, positioned. With few lines of code a sprite sheet is loaded and cached, making it ready for usage.
Actions	Actions are predefined behaviors, like moving a sprite in a defined path over time. Actions can be started with a single line of code.
Sounds	It takes one line of code to play a sound or a background music file. Background music can be stopped at any time with another line of code.

Particles	Cocos2d offers a complex particle system with many parameters like the particle emission rate, color, velocity, size of the particles, etc. Despite the complexity of particle systems and how easy it is to use them, the framework can handle 10000 particles of different colors with a size of four pixels at about 26 fps. Commonly used effects like fire, snow, rain, etc. are already available in the framework.
Labels	Text can be rendered in the screen at a user defined position and with the user selected font with a single line of code.

5.2.3 Box2d Physics Engine

Box2d is a free open source 2D physics engine written in C++. It is one of the two main physics engines with support for iOS, the other one is *Chipmunk*. Box2d offers more functionality, has been around for a longer period of time and is more established than Chipmunk. As for performance, the community seems to agree in that Box2d achieves as good performance results as Chipmunk, or better. It is highly portable, it does not use any STL container and it can be compiled and linked with CMake. It comes with a testbed with code samples covering the whole API's functionality. Some of the engine's features:

- Continuous collision detection. This is used to solve the problem of objects traversing each other when moving fast. It is straightforward to prevent such a situation between two objects. But correctly (and efficiently) calculating collision values for `n` objects requires state of the art algorithms.
- Island solution and sleep management. Optimization that prevents groups of objects that are resting (their speed is zero) and are only colliding against other objects at rest from participating of the simulation.
- Collision groups and categories. By default, the engine generates events whenever two objects collide. With box2d, the programmer can define groups of objects and configure which group collides with which other group.

5.3 Subsystem Decomposition

In this section we introduce the initial decomposition of the system and describe the responsibilities of each subsystem. The Model View Controller architectural style is chosen to decouple entity objects from boundary objects. This design decision is directly derived from supportability requirement 2 in (4.5.3).

The `World` Subsystem is the *Model* of the MVC architectural style. The `World` class and every application object are part of the `World` Subsystem. The structure of the application is entirely contained in the `World` Subsystem. This Subsystem can be reused in any other application since it is isolated from every editing functionality.

`Simulation`, `Edition` and `Code Generation` Subsystems are *Controllers*. The `Edition` and `Simulation` Subsystems alter model data after processing user interaction. They also keep the corresponding Views up to date when model objects change.

The `Code Generation` Subsystem does not alter the model, it only uses it and presents a different View.

`Scene`, `Tab View` and `Code View` Subsystems are the *Views*. The `Scene` is used by `Edition` and `Simulation` Subsystems for low level rendering. `Tab View` corresponds to the palette. It is used by the `Edition` Subsystem to add objects and edit their properties. The `Code View` is used by `Code Generation` Subsystem to let the user choose the target platform and to display code generation progress.

Figure 5.1 shows the subsystem decomposition. Dashed lines represent interaction by means via observer pattern.

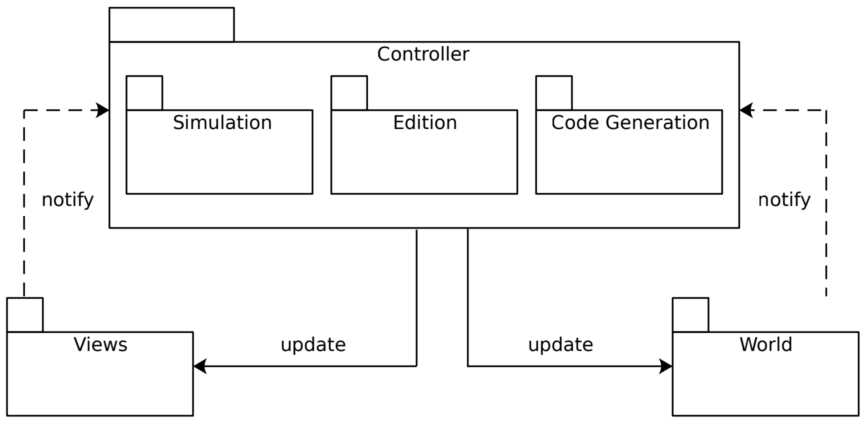


Figure 5.1: System's design as Model View Controller.

5.4 Persistent Data Management

The system requires three kinds of persistent data:

1. Configuration and user preferences data. Examples of this are icon sizes, the width of the palette, the factor by which objects are scaled with the pinch gesture, default number of parts of the rope, etc. Some of these values are constant like the icon sizes, but some can be modified by the user, like the default gravity values.

2. Application state, which consists of every object composing the application, their properties, and the general application settings like the background image.

3. Prototypes (palette items) created by the user. The same user defined palette items may be used in different applications. The palette item's label and the models needed to instantiate the different application objects should therefore be stored persistently.

The Core Data object graph management framework is used for the persistent data management.

Chapter 6

Object Design

In this section the subsystems identified in Section 5.3 are explained in more detail.

6.1 World Subsystem

The `World` Subsystem consists of the application objects and the `World` class. The `World` class aggregates the different `EditableObjects`. The `World` class keeps application data consistent by performing the appropriate checks before modifying its state. For example, it checks whether two objects accept joints before joining them.

6.1.1 Memento Classes

Some classes of this subsystem can be altered during the simulation. Because their state should be restored to their state before the simulation started, they implement the Memento pattern. An Objective-C protocol ensures that memento classes implement functionality to generate a state object, and functionality to restore their state from a state object. Figure 6.1 shows `World` and `PhysicalObject` implementing the `MementoInterface` protocol. Figure 6.2 illustrates the Memento taxonomy. Note that `Rope`, `PhysicalObjectComposite`, `ParticleSystems` and `Triggers` can also be altered at simulation time and therefore implement the Memento pattern.

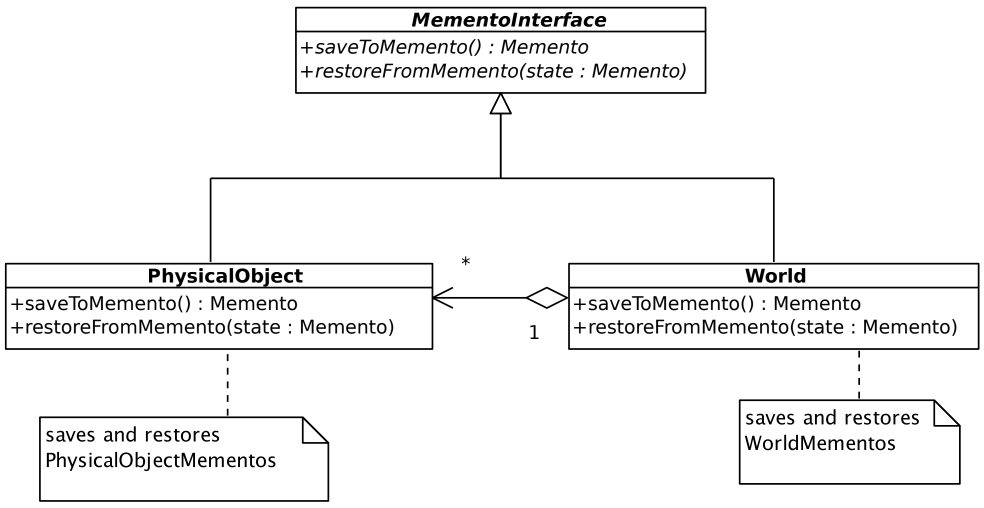

MementoInterface
+saveToMemento() : Memento
+restoreFromMemento(state : Memento)
PhysicalObject
+saveToMemento() : Memento
+restoreFromMemento(state : Memento)
*
1
World
+saveToMemento() : Memento
+restoreFromMemento(state : Memento)
saves and restores PhysicalObjectMementos
saves and restores WorldMementos

Figure 6.1: Structure of Memento classes.

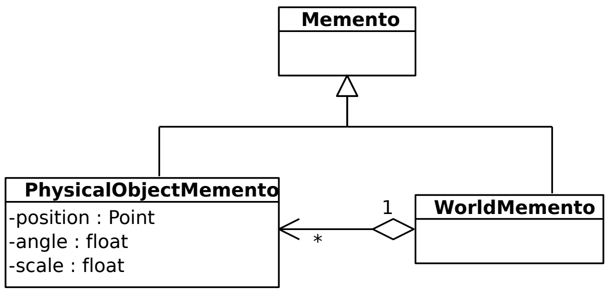


Figure 6.2: The Memento taxonomy.

6.1.2 Connectable Objects

Some `EditableObjects` can be visually connected to other objects. This is illustrated in Figure 6.3 where a `TimeTrigger` is being connected to a `CreateAction`. In order to do this, the user holds the "*Hold to connect*" button in the palette and draws with his finger a line from one object to another. This is the case for example with triggers that are linked to actions, or animation objects that are associated to the object they should animate. Objects that can be interactively connected, inherit from the `ConnectableObject` class, which is a subclass of `EditableObject`.

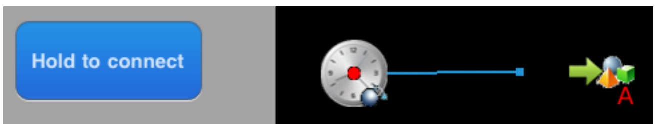


Figure 6.3: A trigger being connected to an action.

6.1.3 Object State Triggers

`CollisionTrigger`, `PositionTrigger`, and `ScaleTrigger` share similar behavior in that they observe objects' state. They implement therefore functionality to add and remove objects and to evaluate whether they should trigger when objects' state change. These triggers inherit from the `ObjectStateTrigger.`This is illustrated in Figure 6.4.

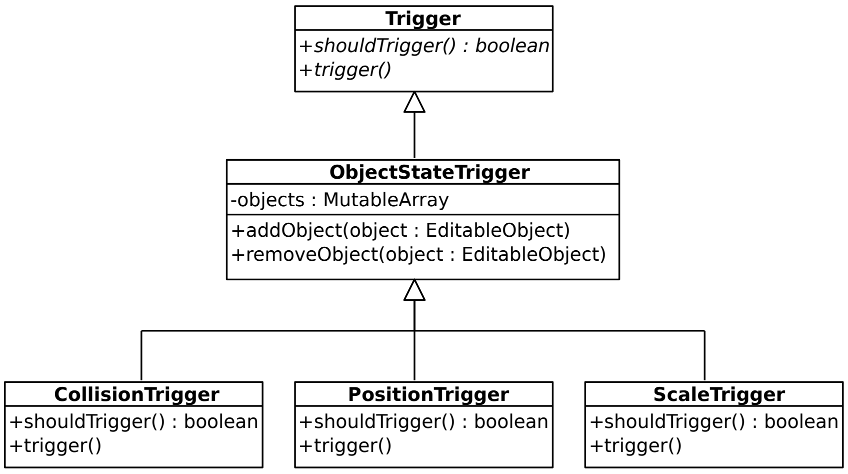

Trigger
+shouldTrigger() : boolean
+trigger()
ObjectStateTrigger
-objects : MutableArray
+addObject(object : EditableObject)
+removeObject(object : EditableObject)
CollisionTrigger
+shouldTrigger() : boolean
+trigger()
PositionTrigger
+shouldTrigger() : boolean
+trigger()
ScaleTrigger
+shouldTrigger() : boolean
+trigger()

Figure 6.4: The `ObjectStateTrigger` taxonomy.

6.1.4 Rope as Proxy Pattern

The `Rope` consists of many elements joined together, and transforming them from the editor once they are integrated into the Box2d physics engine implies destroying the objects and the joints and creating them again with a different position or scale or angle. This violates performance requirement 1 in 4.5.2 (the FPS drop below 30) when the user quickly transforms the rope object for example by moving it around. In order to avoid this problem, the Proxy pattern is used. A `RopeProxy` class is introduced. When the simulation starts, the proxy creates a `RealRope` and when it finishes it destroys it. Clients interact with the proxy and the proxy forwards the method calls to the real rope, if it has already been created. This is illustrated by figure 6.5.

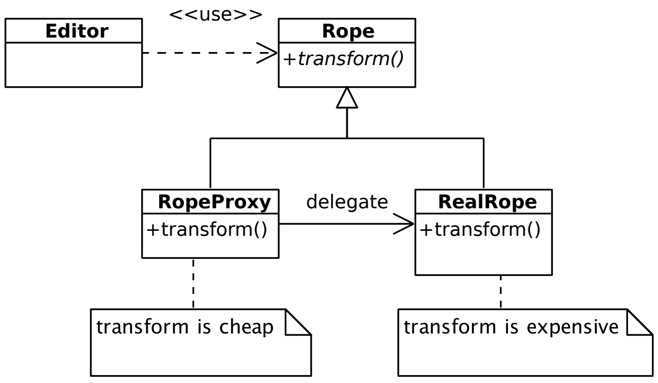


Figure 6.5: Rope as a Proxy pattern

6.1.5 Particle System as Strategy Pattern

The `ParticleSystem` has two different implementations: `LCParticleSystem` and `NormalParticleSystem`. What implementation is used, is decided at runtime, depending on whether the particles should interact with each other or not. If particles should interact with each other, the `LCParticleSystem` implementation is the most efficient. If particles should not interact with each other, the `NormalParticleSystem` is the most efficient. More about the Particle System is provided in Chapter 7.

6.2 Edition Subsystem

6.2.1 Overview

This subsystem has the editing functionality: changing object's properties, transforming objects, adding sprites to them, defining triggers and actions, etc. It receives user input from the `Scene` (mainly for object transformations) and from the `Palette` (object's properties being edited) and updates the `World` accordingly. It also updates the `Scene` whenever the `World's` state changes, for example, when application objects are modified directly from the `Palette`.

The `Edition` Subsystem is composed by the `Editor`, which contains a `Palette`. The Editor was already introduced in the analysis model. The `Palette` handles user interaction with the palette, like functionality to switch tabs.

The relation between the `Editor`, the `Palette` and the `Tab` taxonomy is shown in Figure 6.7.

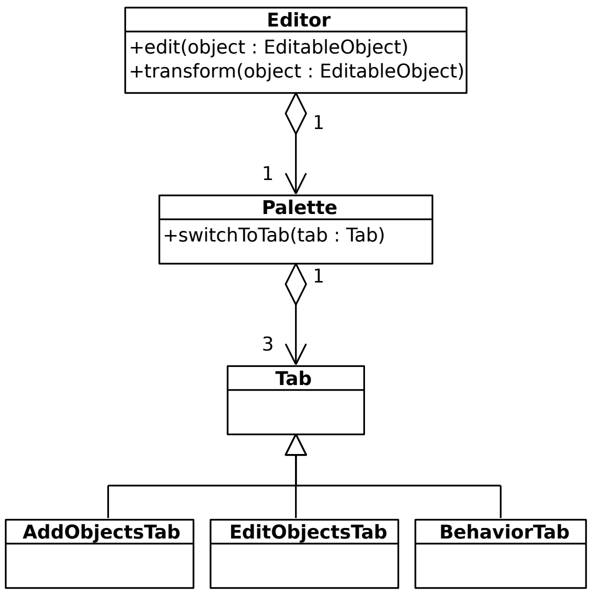


Figure 6.6: Overview of the `Edition` Subsystem.

6.2.2 Palette

The `Palette` is composed by three Tabs: the `AddObjectsTab`, the `EditObjectsTab`, and the `BehaviorTab`.

6.2.2.1 AddObjectsTab

The `AddObjectsTab` is used for dragging objects from the palette to the scene. The `PaletteItem` taxonomy is used to instantiate application objects when added to the scene. The structure of the `AddObjectsTab` is illustrated in Figure 6.7. Note that not every subclass of `PaletteItem` is displayed, for the sake of simplicity. There is approximately one `PaletteItem` subclass for each application object.

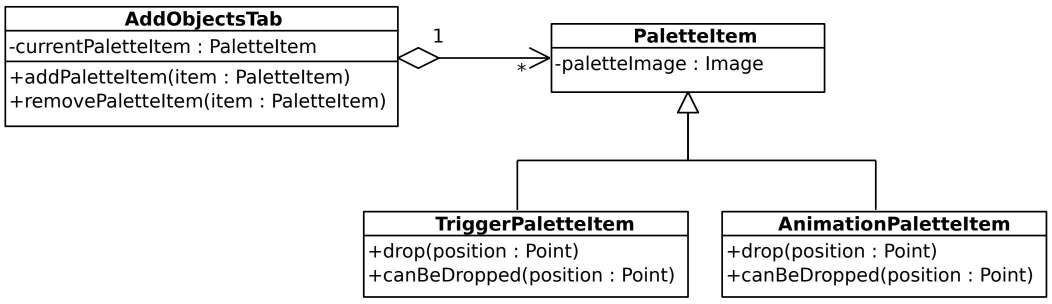

AddObjectsTab
-currentPaletteItem : PaletteItem
+addPaletteItem(item : PaletteItem)
+removePaletteItem(item : PaletteItem)
1
*
PaletteItem
-paletteImage : Image
TriggerPaletteItem
+drop(position : Point)
+canBeDropped(position : Point)
AnimationPaletteItem
+drop(position : Point)
+canBeDropped(position : Point)

Figure 6.7: The `AddObjectsTab` tab.

6.2.2.2 EditObjectsTab

The `EditObjectsTab` is used for editing objects' properties, once they were already added to the scene. `EditingViewControllers` modify model objects after user interaction, and update the GUI to adapt it to changes in the model. This is illustrated in Figure 6.8. Note that not every subclass of `EditingViewController` is displayed, for the sake of simplicity. There is approximately one `EditingViewController` subclass for each application object.

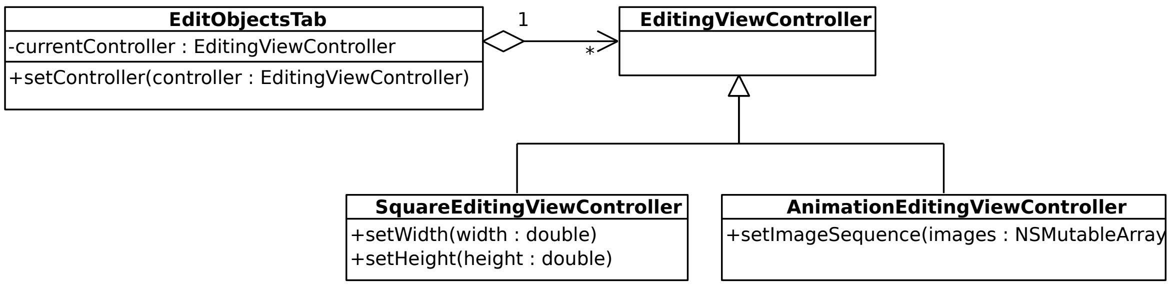
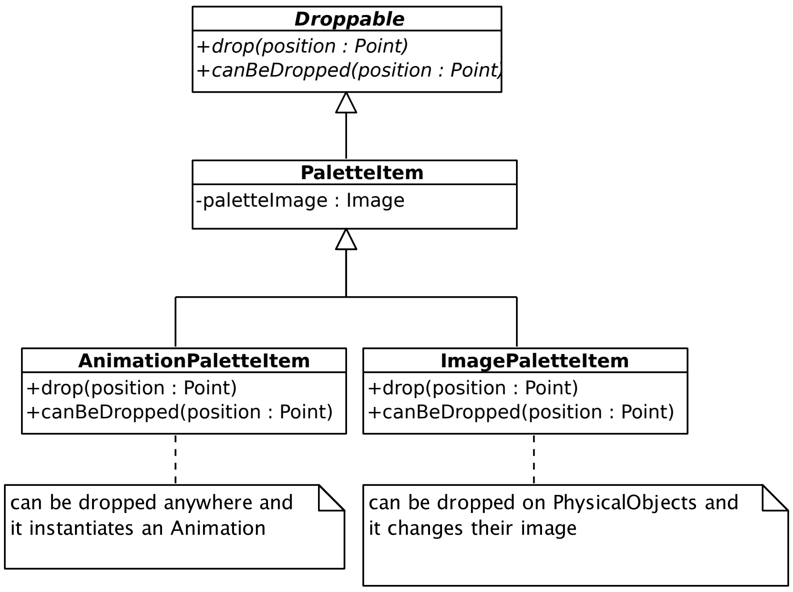

EditObjectsTab
-currentController : EditingViewController
+setController(controller : EditingViewController)
1
*
EditingViewController
SquareEditingViewController
+setWidth(width : double)
+setHeight(height : double)
AnimationEditingViewController
+setImageSequence(images : NSMutableArray

Figure 6.8: The `EditObjectsTab` tab.

6.2.2.3 Droppable Protocol

`PaletteItems` conform to the `Droppable` protocol. `Droppables` are objects that can be dragged into the `Scene` and dropped in certain places. Most `PaletteItems` can be dropped anywhere, and when dropped they create an instance of an application object. For example the `AnimationPaletteItem` can be dropped anywhere in the scene and it instantiates an `Animation` object. `ImagePaletteItems` can only be dropped on already added physical objects, and they do not instantiate a new object, they just change the `PhysicalObject`'s sprite. This is illustrated illustrated in Figure 6.9.



Droppable
+drop(position : Point)
+canBeDropped(position : Point)
PaletteItem
-paletteImage : Image
AnimationPaletteItem
+drop(position : Point)
+canBeDropped(position : Point)
ImagePaletteItem
+drop(position : Point)
+canBeDropped(position : Point)
can be dropped anywhere and it instantiates an Animation
can be dropped on PhysicalObjects and it changes their image

Figure 6.9: The `Droppable` protocol

6.2.2.4 Custom Palette Items

The `CustomPaletteItem` is different to the other palette items, in that it does not instantiate an application object by itself, but it uses other palette items for this purpose. Because the user can group physical objects as a composite object and make it a palette item, the `CustomPaletteItem` also needs to be modeled as a composite pattern. This is shown in Figure 6.10.

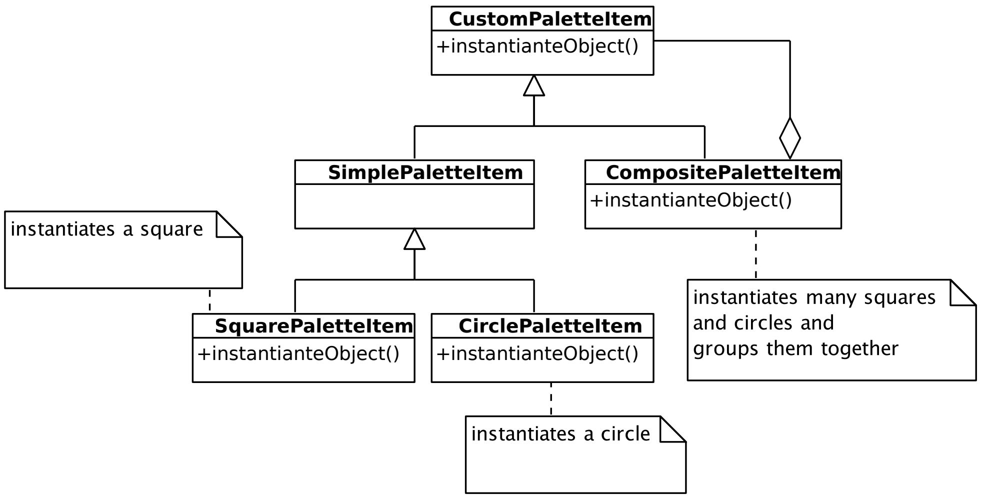


Figure 6.10: The composite `CustomPaletteItem`

Algorithm 6.1 `drop` method of a `CustomPaletteItem`

```
drop: position {
    for every object in parts do
        object.drop:position
    od

    world.group:parts
}
```

When a `CompositePaletteItem` is dropped, the drop methods of every other palette item it is composed by are invoked recursively. After the objects are created, they need to be grouped. Algorithm 6.1 shows the recursive procedure in pseudo code. The group method is directly supported by the `World`.

6.2.3 Dynamic Behavior

The `Editor` observes the world and reacts to some events like when objects are added, removed, or a physical object's sprite has been changed, and performs the necessary changes to the `Scene`. An example is shown in Figure 6.11 as a sequence diagram.

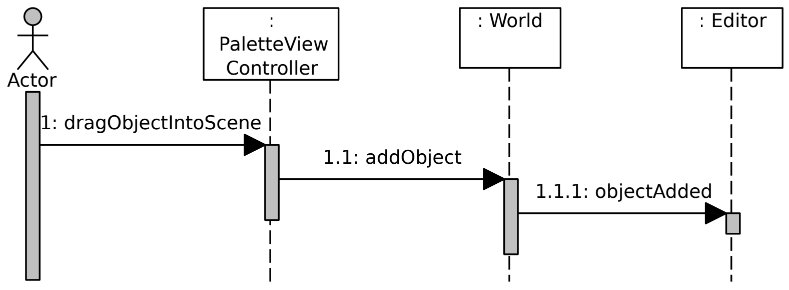


Figure 6.11: Sequence of events after the user adds an object from the palette into the scene.

The `Editor` also reacts to move, tap, pinch and rotate gestures mainly for transforming objects:

- Move gesture moves the selected objects. If no object is selected, the background is moved.
- The tap gesture selects objects. If two objects are overlapping, they are joined. If no object is selected by the tap gesture, every selected object is unselected.
- The pinch gesture scales physical objects. If no physical object is selected, the Scene is scaled, and as a consequence, every object in the scene.

- A double tap gesture restores the background and every object to their original scale. This does not affect object's scale set with the pinch gesture *on* the object. It only sets back the effects of a pinch gesture on the Scene.
- A rotate gesture rotates the selected physical object.

`PaletteViewController` handles the events when the user starts a move gesture on some `PaletteItem` or when a `SoundPaletteItem` is tapped, in which case the sound is played. `PaletteViewController` is implemented as a State design pattern. When an object is first tapped, it is set as the current palette object. Because every palette object implements the `Droppable` protocol, the palette does not need to know what is the current palette object. When the current object is released, its `drop` method is invoked, which is implemented differently by each `PaletteItem` subclass.

There is approximately one `EditingViewController` class for each application object. These classes are used to edit application objects' properties. They react to user interface events and update model objects directly, and sometimes also through the `Editor`. For example, physical objects properties like `friction` are directly changed from the object. But adding an `Action` to a `Trigger`, or connecting an `Animation` to a `PhysicalObject` is done through the `Editor`. The `Editor`'s state changes so that it does not react to the gestures normally, but it allows the user to connect two objects and notifies the `Scene`, which provides visual feedback of the connection.

6.3 Simulation Subsystem

The `Simulation` Subsystem implements the necessary functionality to start and stop the application, and to handle user interaction during the simulation. It consists mainly of the `Simulator` class. It has the following responsibilities:

- Storing / restoring `World's` state when the simulation starts / ends.
- Notifying the `World` that the simulation has started.
- Notifying the `World` of application updates.
- Handling user interaction. By default, users can move objects with their fingers and cut ropes with a swipe movement. The `Simulator` detects this gestures by processing input received from the `Scene` and tells the `World` to perform the necessary actions.

6.4 Code Generation Subsystem

The `Code Generation` Subsystem consists of the `CodeGenerator` taxonomy and the `Template` class. The `CodeGenerator` uses the `Template` class. `CodeGenerator` is implemented using the Strategy Pattern. The policy is integrated into the `CodeGenerator`. The `Code Generation` View tells the `CodeGenerator` what strategy should be used. The `Template` represents a code template and implements functionality to load and save the external code file. The `Code Generation` Subsystem is illustrated in Figure 6.12.

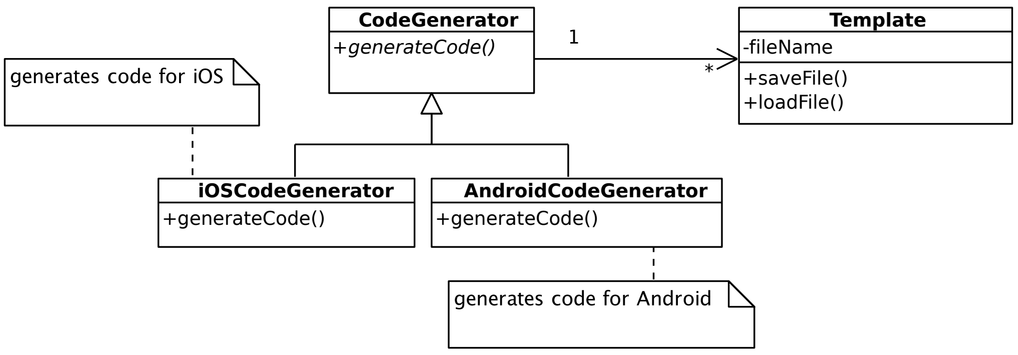


Figure 6.12: Code Generation Subsystem.

6.5 View Subsystems

The `Scene` Subsystem consists only of a `Scene` class. It is used by `Editor` and `Simulator` for low level OpenGL based rendering. Most of its functionality is implemented in the Cocos2d Framework.

The `Tab View` and the `Code Generation` Subsystems are composed of different *xib* GUI files, designed with the interface builder integrated in XCode. Figure 6.13 illustrates how a GUI file looks like viewed with XCode.

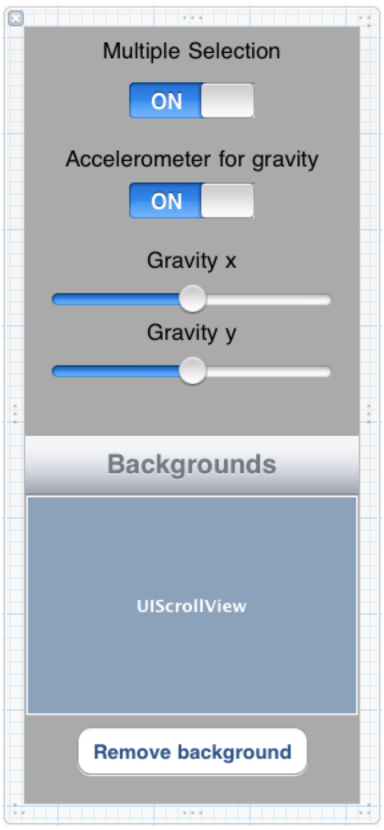


Figure 6.13: General application parameters' user interface file in XCode.

Chapter 7

Particle Simulation

Particle systems are often used in games to simulate certain fuzzy phenomena like fire, clouds, rain, snow, etc. Figure 7.1 shows a fire simulation produced with the Cocos2d particle system.

Particle systems have similar problematics to the ones that originated the current work: it is hard to predict their outcome at compile time. This causes developers to constantly test if they achieve the desired effects.

A particle system is integrated in the system, with the purpose of interactively testing its different parameters without the need to code and run the application each time some parameter needs to be tested.

Implementing a particle system mainly consists of repeatedly updating particles' positions, velocity and other particle values, and rendering. An update iteration is shown in Algorithm 7.1. Algorithm 7.2 shows how particle values are updated.

Updating the particles decreases their life and radius, and updates their color as an interpolation between the particle's initial and end color. The life of the particle is used for the interpolation such that when the particle is initially created, its color matches the initial color and the alpha value is 255, and when the particle dies its color matches the end color and its alpha value is 0. If the particle's radius is decreased to a negative value, the particle's life is set to -1. Particles with negative life are not updated or rendered.

A simple rendering technique is to draw a filled circle for each particle. The Co-

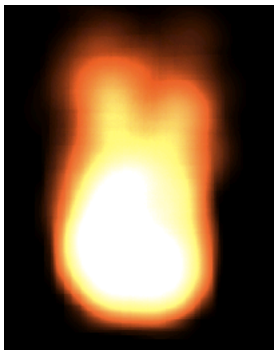


Figure 7.1: Fire effect using a particle systems.

Algorithm 7.1 Particles system main update

```
-(void) update:(float) dt{
    [self compX:dt];
    [self compV:dt];
    [self updateParticles:dt];
}
```

Algorithm 7.2 Particles update

```
-(void) updateParticles:(float) dt{
    for (int i = 0 ; i < currentNumParticles; i++) {
        Particle * particle = &particles[i];
        if(particle->life > 0){
            particle->life -= vanishingSpeed * dt;
            particle->radius -= diminishingSpeed * dt;
            double lifed = (initialLife - particle->life) /
                                        initialLife;
            Color c1 = particle->color1;
            Color c2 = particle->color2;
            particle->color.r = (1-lifed) * c1.r + lifed * c2.r;
            particle->color.g = (1-lifed) * c1.g + lifed * c2.g;
            particle->color.b = (1-lifed) * c1.b + lifed * c2.b;
            particle->color.a = (1-lifed) * 255;
            if(particle->radius < 0)
                particle->life = -1;
        }
    }
}
```

cos2d's ccDrawCircle is used for this purpose. It approximates a circle with triangles. The number of triangles used can be adjusted. For particles with sizes as normally needed by particle systems, seven triangles are enough.

Some considerations about the experiments for which results are shown in this chapter:

- For the tests, the particle system was simplified. The color, radius, life, etc. attributes have been removed from the particle data structure. The only particle attributes used were position, velocity and force.
- Periodic boundary conditions were used. This means that particles exiting the domain from any side re-enter the domain from the opposite side. It has been chosen this way so that the number of particles on screen remains constant, and thus the workload remains also constant.
- The maximum FPS over a period of time were used as performance metrics.
- All the tests were conducted on an iPad 1.

7.1 The Linked-Cell method

One of the features of the particle system makes particles interact with each other. This gives place to a completely different way of setting the particles' movement pattern, and is at the same time easy to configure. A potential function is defined that determines the force exerted on two particles, depending on their distance. This kind of feature can not be found in classical particle systems. Classical particle systems simply set particles' initial velocity to some value and update it. The reason is simple, making particles interact with each other means running a $O(N^2)$ algorithm: for every particle, its distance and potential to every other particle has to be calculated.

A typical potential function is the Lennard-Jones potential: $P(d) = a\left[\left(\frac{\sigma}{d}\right)^{12} - \left(\frac{\sigma}{d}\right)^{6}\right]$. The potential is parametrized by σ and a. a determines the depth of the potential. A big a makes the negative curve steeper. σ determines the zero crossing of the potential. 7.2 illustrates the potential for a = 10 and $\sigma = 1$. In this particle system, negative potentials result in attraction forces, positive potentials result in repulsion forces.

One possible optimization to the $O(N^2)$ problem consists on exploiting the anti-symmetry property of the force between two points. If the potential between particles A and B is calculated, and particle A gets a force of F, then particle B should get a force of -F. This saves computing distance and potential for half of the particles, making the complexity $O(N^2/2)$. But the problem remains $O(N^2)$ asymptotically.

The Linked Cell (LC) method is one of the oldest algorithms in molecular dynamics, used to reduce the complexity of such a problem to $O(N)$. It is based on the observation that forces decay very rapidly with the distance between particles. So ignoring attraction forces when particles have a distance bigger than the so called `cutoff` radius does almost not alter the simulation results. A naive implementation consists of measuring distances between particles and computing potentials only "if

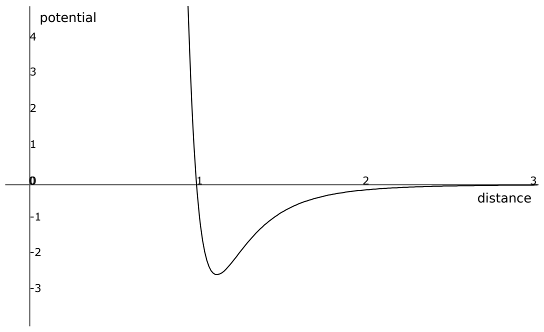


Figure 7.2: Adding a sprite to an object.

1000 particles	FPS
Normal	-
Linked Cell 4x4	7
Linked Cell 16x16	47
Linked Cell 32x32	53
Linked Cell 64x64	47
Linked Cell 128x128	29
Linked Cell 256x256	11

Table 7.1: Comparison between a normal particle system implementation and the linked cell optimization.

distance is smaller than `cutoff`". This saves some computation time, since potential is now only calculated for a few particles. But the complexity remains the same.

In order to achieve a complexity of $O(N)$, only the particles inside the cutoff radius of each particle should be iterated. In order to identify the particles inside the cutoff radius of a particle, new data structures are needed. The approach consists on splitting the domain in a grid. Each cell of the grid stores the particles within the cell in a linked list data structure. When the potential has to be calculated for a particle, only the particles in the neighbor cells need to be iterated. This is illustrated by Figure 7.2. If the potential is calculated for the red particle, only the cells in light grey have to be checked. The cutoff radius is shown in a dark grey color. This does not only identify the particles inside the cutoff radius, but other particles as well. So refining the grid allows for iteration of less particles, since the cutoff radius is better approximated by the grid. Table 7.1 compares the normal implementation of a particle system with the linked cell implementation for 1000 particles.

This optimization is not for free. Working with linked lists limits the pre-fetching done by the processor, and therefore the efficient use of caches. This is because processors are optimized for sequential loops that access contiguous memory. Memory is loaded sequentially in blocks. Because when using linked lists memory is not accessed sequentially, memory blocks are often removed from the cache before they can be reused. Additionally, the data structures need to be kept up to date. Whenever a particle traverses an edge in the grid, it has to be removed from the linked list of its previous cell, and added to the linked list of the cell it entered.

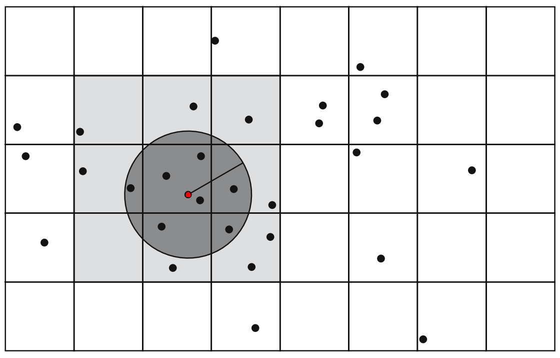


Figure 7.3: Linked Cell method.

10000 particles	FPS
Normal	55
Linked Cell 16x16	55
Linked Cell 64x64	55
Linked Cell 256x256	54
Linked Cell 512x512	26

Table 7.2: How the number of LC cells affects performance.

The more cells used in the LC algorithm, the best the cutoff radius can be approximated, and therefore the fewer the particles that need to be iterated. So increasing the amount of cells should increase performance. This is specially the case when there are many particles per cell. When cells are dense, the costs of iterating through them is high. It is therefore worth accurately approximating the cutoff radius in this situation.

More LC cells also increase the amount of memory in use and cause overhead in keeping the particles in the correct cell. Table 7.2 shows how the number of cells used by the LC algorithm affects performance. In this experiment no interaction between particles took place. This is the reason why there is no gain when using LC. In table 7.1, performance gain was observed up to 32x32 cells, but more cells caused the overhead in keeping the data structures up to date to be bigger than the actual gain of the optimization.

7.2 Rendering optimizations

A technique to render the particles was previously described, where each particle is rendered as a filled circle. One of the reasons why this approach is far from being optimal is because sending data to the GPU has some latency time. While data is being transferred through the bus from CPU to GPU, these are idle. OpenGL acknowledges this problematic and offers low level functionality for sending chunks of data. It even lets the application reserve memory in the GPU, making the rendering much faster because no additional memory needs to be reserved in each rendering loop.

Algorithm 7.3 Vertex Buffer Object initialization

```
-(void) initVBO{
   vertices = (Vertex*)malloc( sizeof(Vertex) * numParticles );
    glGenBuffers(1, &verticesID);
    glBindBuffer(GL_ARRAY_BUFFER, verticesID);
   glBufferData(GL_ARRAY_BUFFER, sizeof(PointSprite)
      * numParticles, vertices, GL_DYNAMIC_DRAW);
   glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Algorithm 7.4 Vertex Buffer Object update

```
-(void) updateVBO{
   glBindBuffer(GL_ARRAY_BUFFER, verticesID);
   glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(Vertex)
      * numParticles, vertices);
   glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

System memory is first allocated and mapped to the video device's memory, and is then flushed to the GPU before the rendering takes place. The OpenGL feature that makes this possible is called vertex buffer object or VBO. Algorithm 7.3 shows how the memory for the particles is allocated in system's memory and a vertex buffer is initialized. Algorithm 7.4 shows how the data is flushed to graphics card.

Another important reason why the previous rendering technique is inefficient, is because for each particle different triangles needed to be uploaded to the graphics card and rendered.

One feature of OpenGL that was introduced to OpenGL implementations for particle system rendering are the so called *point sprites.* Point sprites let an application render a texture by specifying a single point. The old way to render textures required specifying four different points, and extra work dealing with projections to make the texture was oriented facing the camera. Point sprites are automatically aligned with the camera.

The vertices stored in the GPU with the VBO technique are transformed into textures directly in the video card. The combination of these two techniques drastically increases the particle system's rendering throughput. The circle-based rendering implementation delivers around 7 FPS for 5000 particles, while the optimized version delivers around 55 FPS for 10000 particles. The test results presented in the previous section were obtained with the optimized rendering implementation.

7.3 Conclusions

Particles interaction is very costly and that is why no particle system implements such a functionality. But it provides a lot of flexibility and is at the same time easy to configure. Users only need to set a single potential function in order to achieve completely different particle behaviors. In the future, the user could be able to create custom potential functions using the screen touch capabilities of the device.

Different optimization techniques have been implemented, tested and their results analyzed. The main purpose of the tests was to show what is the maximum amount of particles that can be simulated with and without particle interaction, and by which factor the LC implementation is more efficient than the normal implementation when particles can interact with each other.

Without particle interaction the tests deliver high frame rates of up to 55 FPS for 10000 particles. The *ParticlesPerformance* test in the Cocos2d testbed, is able to render 10000 particles of the same size at only 25 FPS on the same device.

The Linked cell is compulsory if particles must interact with each other. A simulation of 1000 particles freezes with the normal implementation, while it can achieve 53 FPS if the LC grid size is adequately tuned.

Chapter 8

Evaluation

8.1 Prototypical implementation

Figure 8.1 presents the three tabs used to add the objects to the screen and edit their properties.

The *objects* tab has containers for different kinds of objects: physical objects, images, animations, sounds and music.

The *edition* tab is used for setting the properties of objects. Common GUI elements used are sliders for integer and floating point variables, and switches for boolean attributes. More complex elements are often present, in the form of pop ups to select specific properties, like the images in an animation, the collision flags for a physical object, the colors of the particles in a particle system, or the sprite in an action that changes objects' sprites. These pop up elements are shown in Figure 8.2 and 8.3.

The *triggers and actions* tab uses the same containers as the objects tab, providing an homogeneous look and feel.

8.2 Achievement of Objectives

In this section the implemented prototype will be evaluated with the requirement specification presented in Chapter 4.

8.2.1 Realized Functionality

8.2.1.1 Editing

The prototypical implementation of TangoPhysics fulfills the main functional requirements. Every object identified in the editing functional requirements is supported by the tool.

Thanks to the State design pattern, the `Editor` does not know what object it is editing, and does not need to implement special behavior for each object. Whenever an `EditableObject` is selected by a gesture, it is set as the current object. Because every `EditableObject` responds to user gestures in a different way, the `Editor` does not need to worry about what instance it interacts with. Adding a new object is as simple as creating the new class and a new `PaletteItem` that instantiates the object.

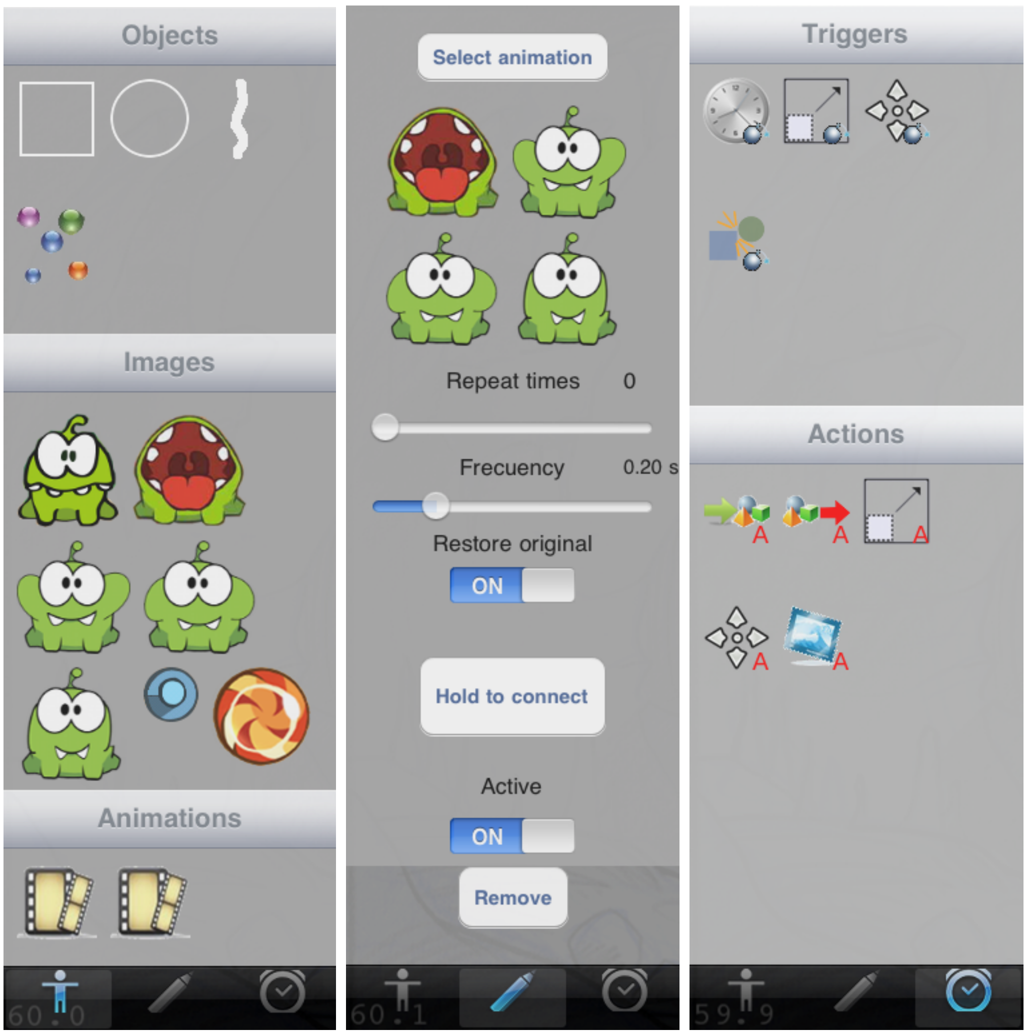
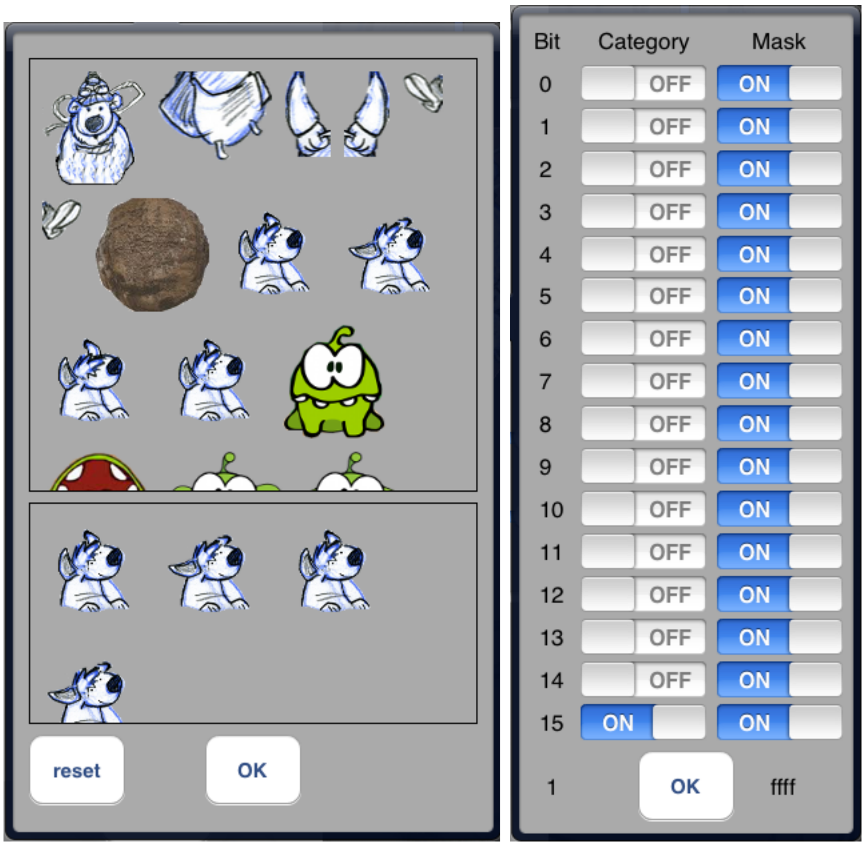
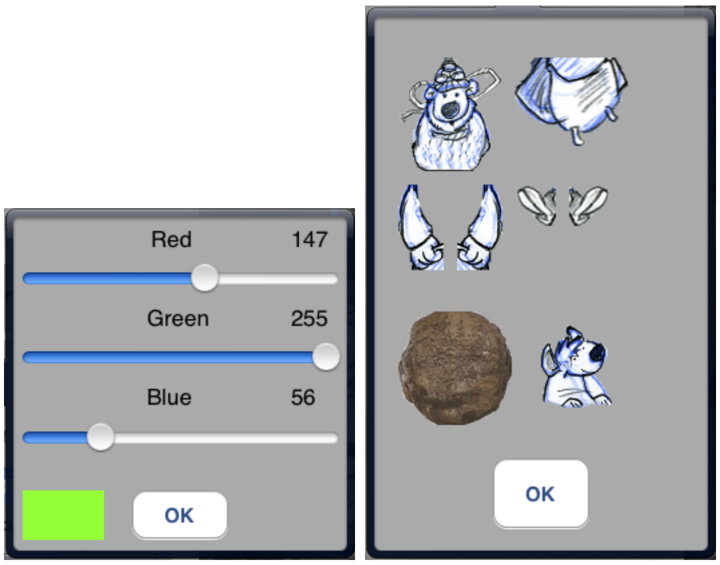


Figure 8.1: The different tabs of the palette, from left to right: the objects tab, the edition tab, and the triggers and actions tab.



Bit	Category	Mask
0	OFF	ON
1	OFF	ON
2	OFF	ON
3	OFF	ON
4	OFF	ON
5	OFF	ON
6	OFF	ON
7	OFF	ON
8	OFF	ON
9	OFF	ON
10	OFF	ON
11	OFF	ON
12	OFF	ON
13	OFF	ON
14	OFF	ON
15	ON	ON
1		ffff

Figure 8.2: Special pop up elements used for selecting complex attributes. From left to right: an animation picker and a collision picker.



Red 147
Green 255
Blue 56

Figure 8.3: Special pop up elements used for selecting complex attributes. From left to right: a color picker and an image picker.

Editing object's properties requires adding a new interface file, which can be done visually in few minutes, and implementing a controller for that interface. Linking the view with the controller is also done graphically thanks to the XCode GUI integration. In most cases, the controller directly modifies the object's properties, resulting in a few lines of code.

Sometimes the `Editor` needs to mediate the properties edition, like when an object needs to be connected to another object. This requires making the `Editor` enter the connection state. The `Editor`, when in editing state, will detect what objects the swipe gesture beings on, and where it ends. It then asks both `EditableObject`s whether they can be connected to each other.

Labeling objects and adding them into the palette is not yet part of the implementation. Labeling objects can be achieved with little effort. It is mainly about adding the label field to the `CompositePhysicalObject`, and adding a text box in the corresponding GUI file. Letting the user drag objects into the palette is also straightforward. An algorithm was already proposed that uses standard palette items to instantiate custom palette items.

Persistency is also not part of the current implementation.

8.2.1.2 Simulating

Every functional requirement related to the simulation has been achieved. The simulation can be started in any moment, and the editing state is stored thanks to the Memento pattern. The only supported gesture to switch between states is currently the *shake* gesture.

8.2.1.3 Generating Code

Code generation is not part of the current implementation.

8.2.2 Nonfunctional Requirements Accomplishment

8.2.2.1 Usability

One of the goals of this system's design was to provide simple interfaces without advanced features and without many objects, that is intuitive and that can produce results quickly. Several factors influence the achievement of this goal.

The user interface is not crowded with high level objects. It displays by default primitive objects. The decision of including more complex objects in the palette is left to the user. Joints and object transformations are not even part of the interface, they are achieved via gestures. This helps in making the interface not only simpler, but also intuitive.

Figure 8.4 illustrates how scaling and rotating objects can be done with other tools. Even if no usability test has been conducted to determine whether actions with TangoPhysics can be done in less time than with other tools, it can be assumed that gesture based transformations require considerably less time than GUI based transformations. The preferred way to scale or rotate objects with existing physics editors is to first select the object, and then use GUI widgets to directly modify the value. It is clearly faster to start scaling or rotating an object with a gesture, for two reasons. First, no additional step is needed to select the object. When the rotation gesture starts on an object, the system knows what object is to be rotated. Second, the user receives feedback while the object is being scaled or rotated. If the user deals directly with the scale or angle values, he can define the values precisely, but only in rare situations he will already know the value he needs. Users typically want to test visually the transformations applied objects, that is the purpose of a physics editor after all.

Some editors like BoxCAD let objects be rotated graphically. The object is first selected, a rotation icon is displayed, and the user must click on it and drag the mouse in some direction. Many users feel more comfortable with this approach compared to the one described above. But the first selection step still has to be done and it is counter intuitive. It is counter intuitive, because the user does not know how the tool maps mouse distance to scale or rotation factors, nor does he know in what direction should the mouse be moved. While some tool may rotate clockwise objects with positive mouse movements in the y axis, and counter clockwise by moving the mouse down, in the negative y axis, some other tool may use another rotation convention, may use the x axis or even diagonal directions. This does not happen when scaling objects with the pinch gesture, or rotating them with the rotation gesture.

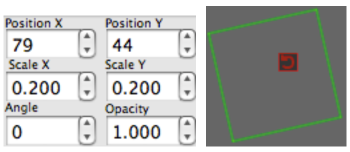
Similar problems occur when moving objects with the mouse. Using the fingers to transform objects is more direct than using mouses, touch pads and keyboards, since users are more familiar their own fingers than to the usage of external devices.

Furthermore, because different tools accomplish object transformations in different ways, as described previously, users will often need to explore the interface, or even read a manual to find out how the transformations can be done with a specific tool. Gestures are on the other hand, the standard transformation procedures in multi-touch devices. They are used by different applications and by the iOS operating system itself. Therefore, any user with minimal experience in iOS is more likely to know beforehand how to transform physical objects using the proposed system, rather than using any of the others.

User familiarity advantages are not only to be appreciated in relation to user gestures. The system was created using usability standard, like the Cocoa framework, the *drag and drop* feature, and the functionality to connect different objects. The Cocoa framework enforces the creation of applications with similar look and feels. This is in contrast to applications created on different operating systems using a wide range of APIs (Qt for C++, Java Swing for Java, X11 for C, etc.). The *drag and drop* functionality was done in the way it is typically done: when an object is dragged, the system provides visual feedback about the places where it can be dropped, as shown in Figure 8.5. This is used in both, adding objects to the scene and to the palette. Connecting objects to each other is also done intuitively dragging a line from the first to the second object.

8.2.2.2 Performance

In the different applications created with the system, including the Rope Game and one level of “Cut the Rope” all performance requirements were achieved, excepting



Position X 79
Position Y 44
Scale X 0.200
Scale Y 0.200
Angle 0
Opacity 1.000

Figure 8.4: GUI functionality for scaling and rotating objects in LevelHelper (left), and for rotating objects in BoxCAD.

Figure 8.5: Adding a sprite to an object.

performance requirement 2 (the simulated application should deliver at least 95% of its potential performance) that has not been tested. However, both applications delivered an almost constant frame rate between 59 and 60 frames per second.

Furthermore, the `Simulation` Subsystem has minimal functionality, mainly related to setting up and launching the simulation. Most of the performance during simulation depends on the `World` Subsystem, that is meant to be reused in created applications. This makes it unlikely for the system to fail to meet requirement 2.

8.2.2.3 Supportability

The three supportability requirements identified in section 4.5 directly influenced the system design. The Model View Controller architecture was chosen providing the system of a set of lowly coupled components easy to replace. The View Subsystems and some part of the Controller Subsystems need to be replaced if the system should run on other platforms. The Model and View Subsystems need to be replaced if the user needs to create applications for another domain, while the editing, simulating and code generating logics (Controller Subsystems) can be reused.

The system also supports different source code output forms, although this functionality was not implemented yet. In the worst case the different programming languages will not share any common structure, making it impossible to use the Template pattern to structure the code generation routine. This is why the code generation was designed as a Strategy pattern. This approach lets extenders support new output formats by creating only one extra class.

8.2.2.4 Robustness

Some special care has been taken during the implementation to prevent aforementioned robustness vulnerabilities. This may happen specially when the application being simulated destroys an object and tries to access it after it was destroyed. This problematic has been prevented by making the `DestroyAction` hide objects instead of destroying them. Another way to fix this would have been effectively destroying it and nullifying all the foreign references to it. While the first approach is inefficient in memory because it keeps in memory an object that is not visible anymore, the second one could cause the simulation to freeze shortly if there are many objects with many references to other objects. Furthermore, keeping the object "alive" lets the application make it appear again with all the same properties, and to even let `Action` objects alter their state while they are hidden. Finally, while editing an application, the developer typically does not add so many objects that it would be necessary to destroy non-visible objects to save memory. Adding many objects and complex time consuming behaviors that can affect performance is normally done programmatically. Once the developer has reached the programming stage, he can destroy hidden objects programmatically if he knows he will not need them anymore, to tune for performance.

The other problematic identified during requirements were time triggers with high frequency. It has not yet been analyzed what time frequencies in connection to what actions can cause application's misbehavior. Once the system is tested with different applications, it will be possible to determine what are the optimal `TimeTrigger's` repetition time limits, and whether warnings should be given to the user in some situations. For now, the user is responsible to ensure the correct application behavior.

While the identified robustness requirements have been fulfilled, there are many other cases that may lead to robustness leaks, that still need to be identified. Every possible combination of objects, triggers and actions needs to be carefully analyzed in order to identify possible leaks.

8.3 Game Creation

8.3.1 Case Study: Creation of the Rope Game

This section presents the implemented prototype and its graphical interface in the process of construction of the Rope Game.

Figure 8.6 shows the initial set up of the game. A few squares and two circles have been added to the scene and their images set. A rope was also added, and the background image set. Emil's rag doll physics have been configured and its joints set. This is achieved within one minute. The background is semi-transparent so it does not affect the visualization of application objects. In simulation time, only the images are shown. Collision polygons (pink shapes) are only shown in the Editor for debugging purposes.

If the device is shaken, the application starts. The ball and the objects can be by default moved with the finger. The ball can be thrown towards the bag. When the ball collides with the character, they both react physically.

A collision trigger is used to define behavior after the ball collides with the bag. It is therefore connected to the bag and to the ball visually. When the ball enters the bag, a sound should be played, and the ball should return to its original position. The collision trigger is connected to a sound object and to a position action. The position action sets the position of the ball to its original position when the collision takes place.

This step done with only three objects and in even less time than the previous

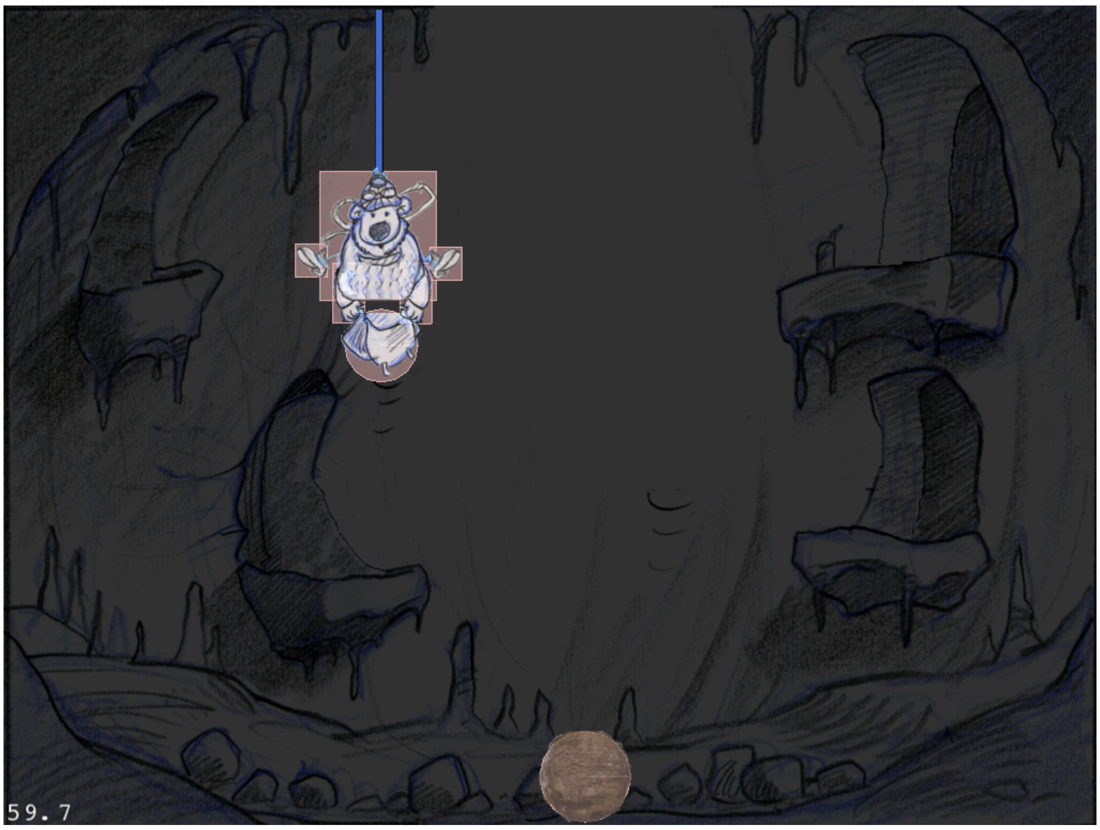


Figure 8.6: Edition of the Rope Game with TangoPhysics (part 1).

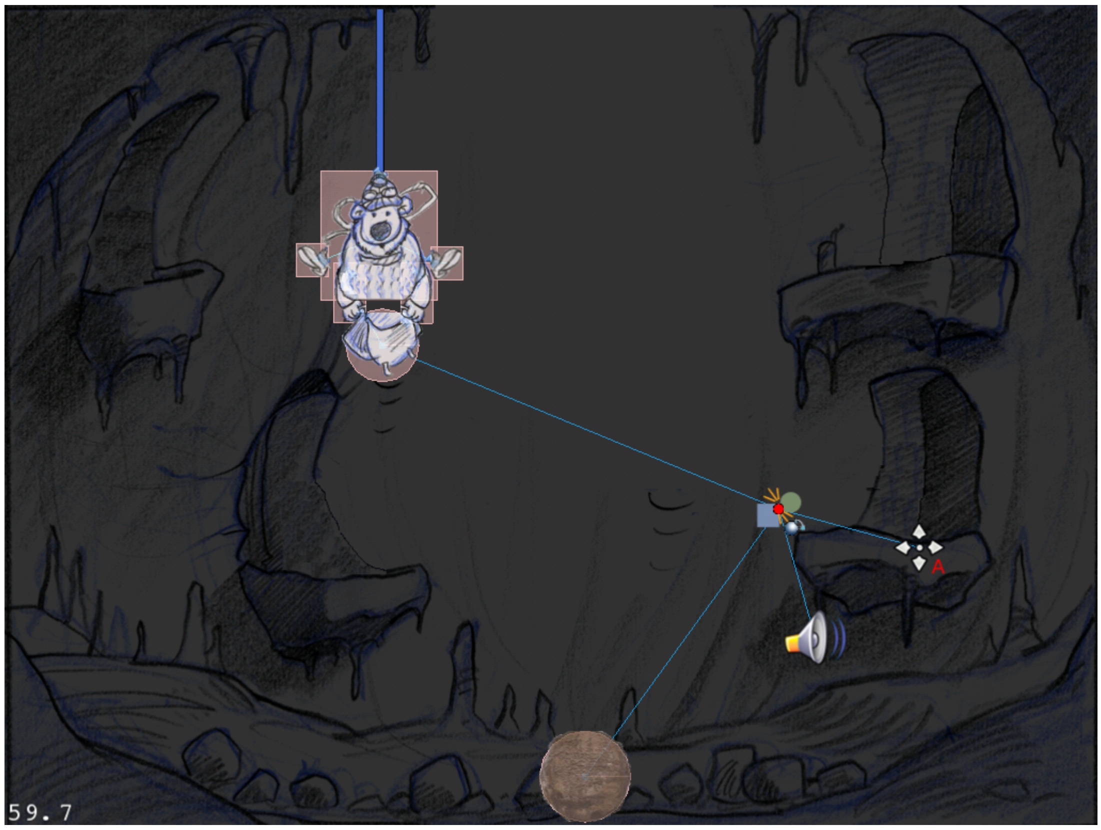


Figure 8.7: Edition of the Rope Game with TangoPhysics (part 2).

step. Figure 8.7 illustrates how the scene under edition looks like.

To add a goblin to a cave, a square object is dragged into the scene and its image set. The goblin should not move, so its type attribute is set to `static`. The `Editor` displays `static` objects in green, to differentiate them from `dynamic` objects that are displayed in pink color.

The goblin should blink the eyes once in a while. A time trigger is therefore connected to an animation object. The animation object is configured to restore the original image and the animation images are selected using an animation picker, as shown in Figure 8.2. The goblin should not collide with any other object in the scene, so the collision picker pop up is used to set its collision flags to 0. All the elements in the screen are shown up to this step in Figure 8.8.

When the ball enters the bag, the goblin should applaud. The collision trigger created in a previous step is then connected to a new animation object, and the animation object is connected to the goblin.

These three objects are added within one minute. Background music can be simply added by dragging the corresponding object.

The created prototype is shown in Figure 8.9. Adding the second rag doll, will be much easier once objects can be added into the palette. It could be done by dragging the already added rag doll into the palette to create a new prototype, instantiating the prototype again, and setting its sprites. In the current implementation, the second rag doll and the triggers and actions have to be created in the same way the first one

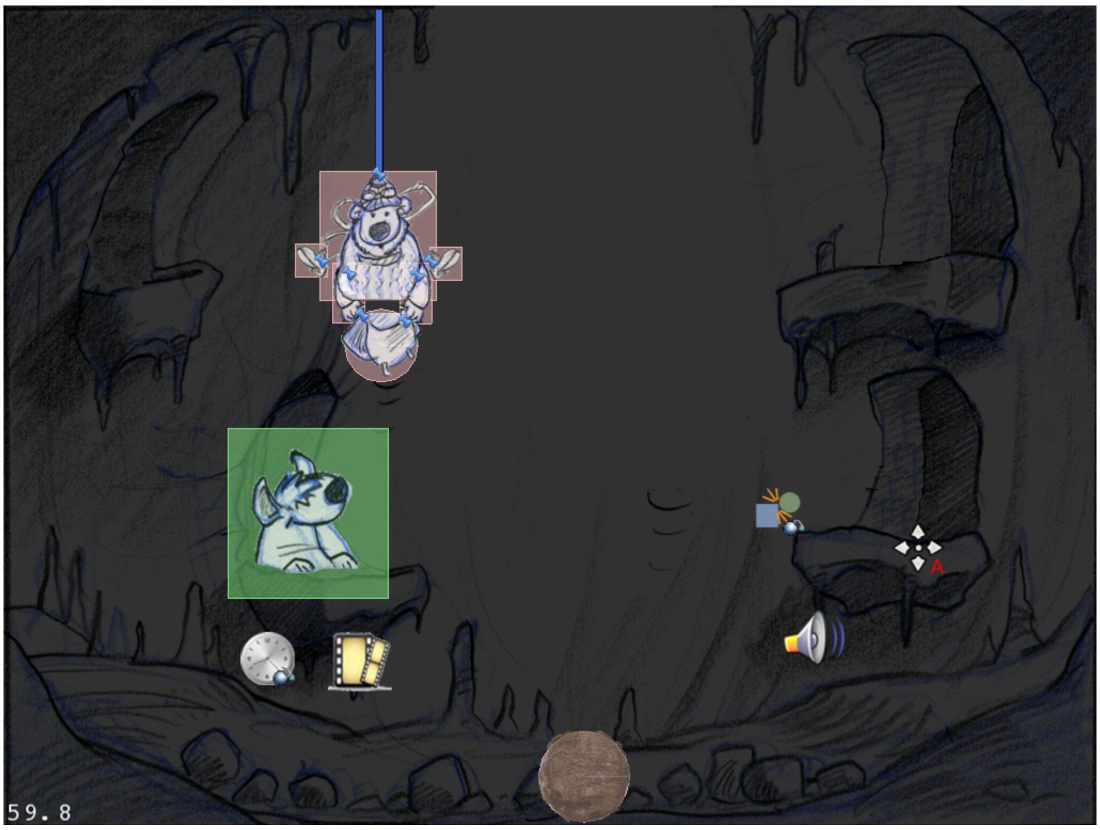


Figure 8.8: Edition of the Rope Game with TangoPhysics (part 3).

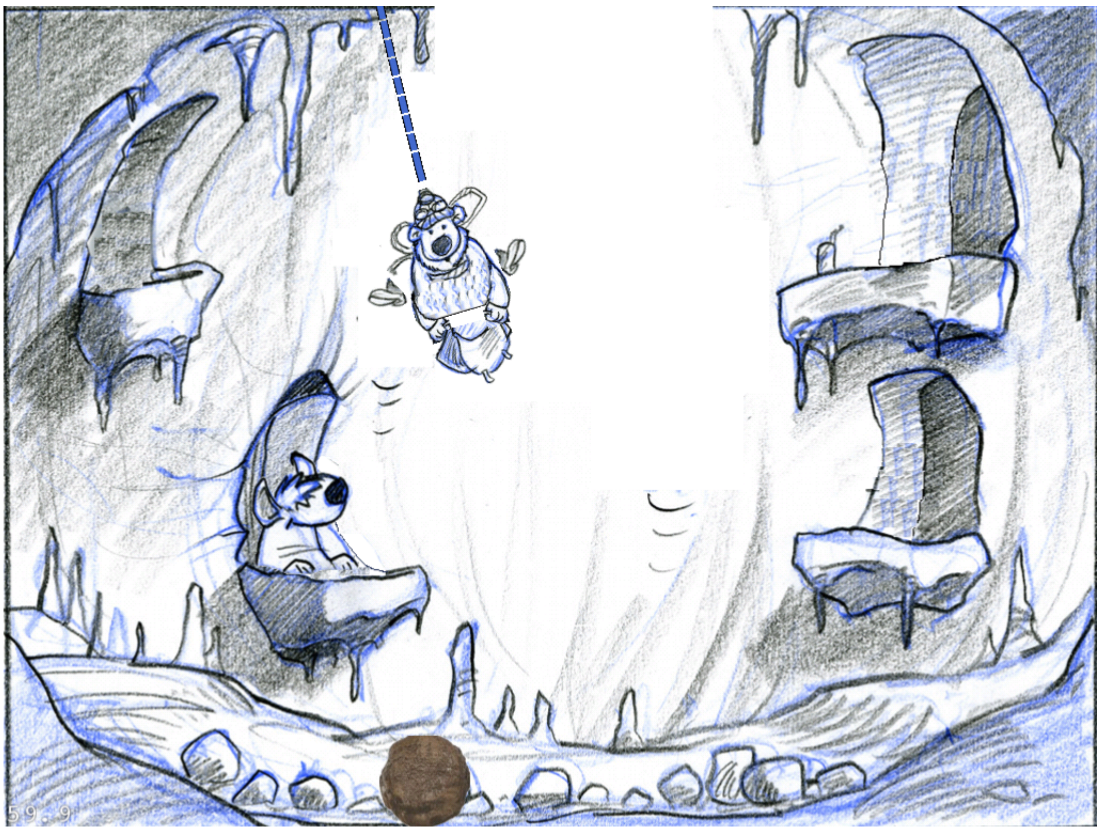


Figure 8.9: Edition of the Rope Game with TangoPhysics (part 4).

was created.

8.3.2 Creation of "Cut the Rope"

Because TangoPhysics was created as an abstraction of the Rope Game, another application was needed to test whether the models used by the tool were abstract enough to enable the creation of other applications. The test was successful; the main functionality of one of the most popular iOS games, "Cut the Rope", could be realized with little effort.

Most functionality needed for the creation of this second game was already provided by the system: physics simulation, support for sprites, sounds and animations, basic behaviors like making objects disappear, randomly animating objects, etc. The only functionality missing was making ropes cuttable, and detecting and displaying the cutting effects upon swipe gestures, and was provided programmatically.

Figure 8.10 illustrates the edition of a level of the game, and Figure 8.11 shows the results. When the first rope is cut, the candy touches the star, a sound is played, the star disappears and the first star in the score (top left of the screen) is filled. The player waits for the right moment and cuts the remaining rope. If the candy hits the monster, an animation is displayed where the monster eats the candy, a sound of the monster eating is played and the candy disappears.

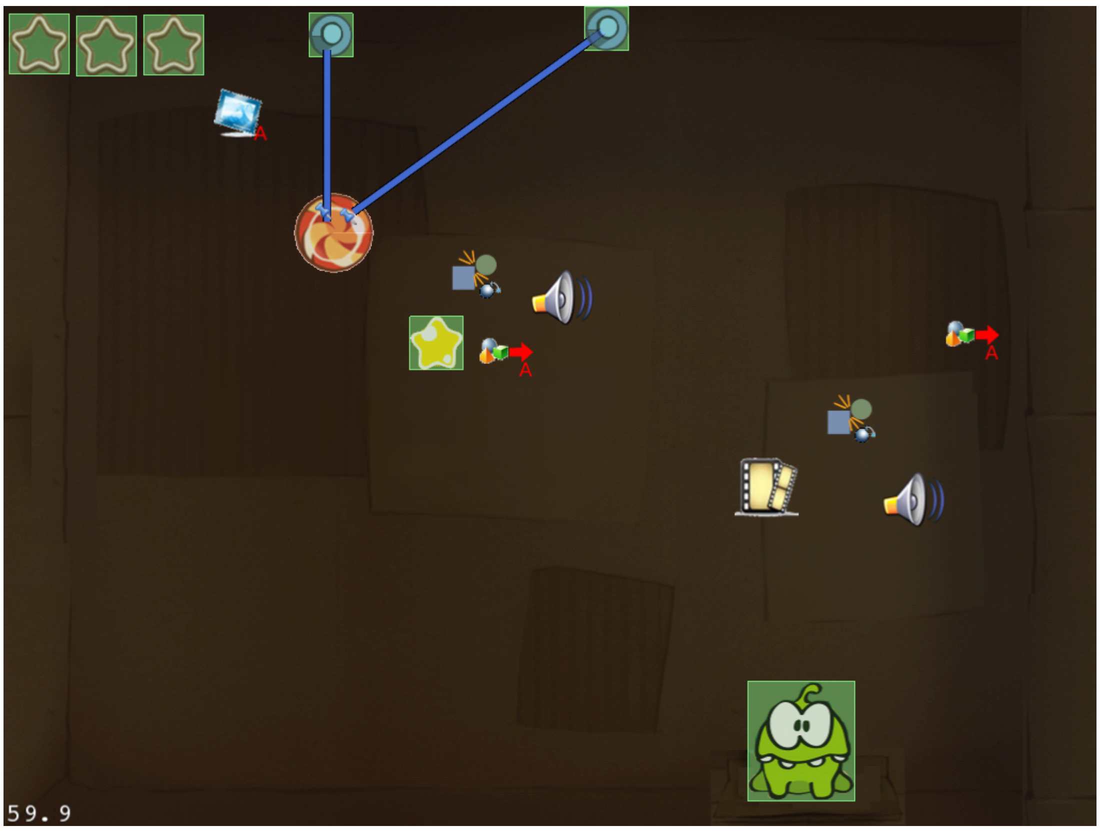


Figure 8.10: Edition of “Cut the Rope” with TangoPhysics.

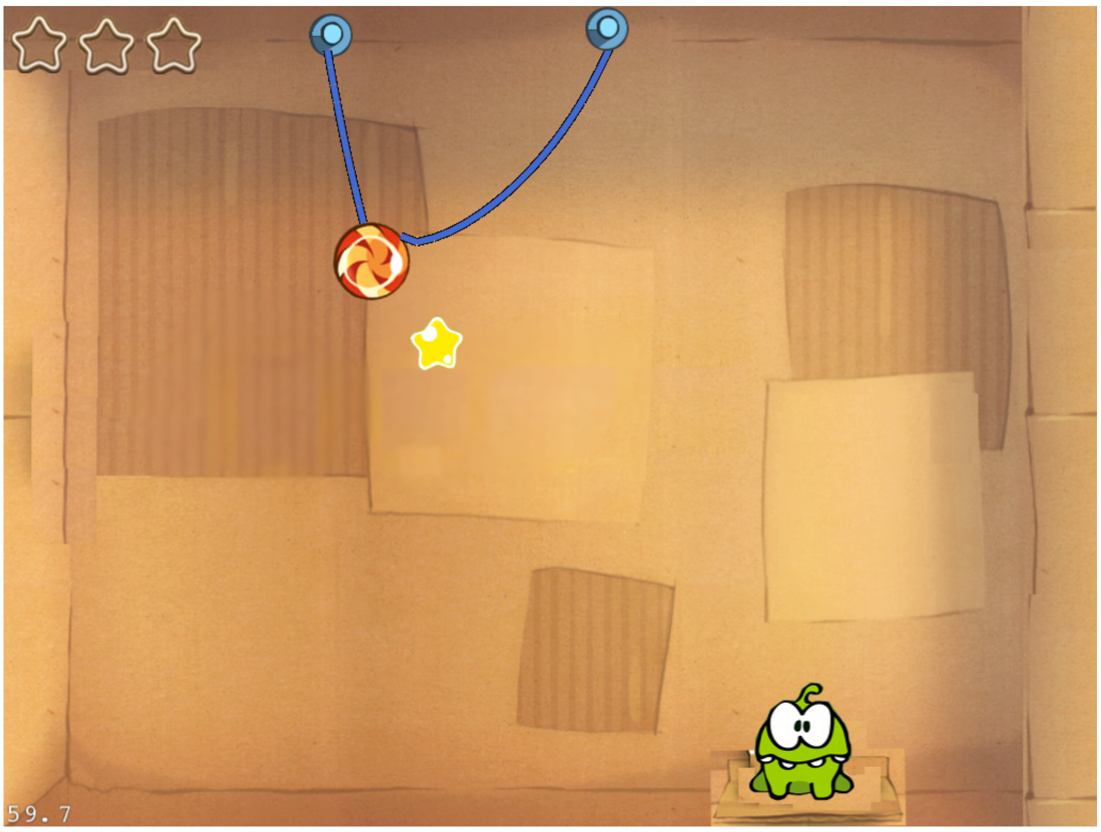


Figure 8.11: Prototype of “Cut the Rope” created with TangoPhysics.

Chapter 9

Conclusions and Future Work

The main objectives of the project were accomplished. While the RopeGame was developed in three months of full time work, TangoPhysics can be used to create a very complete prototype in only five minutes. The system also proved to be flexible enough to allow the creation of one of the currently most popular iOS games, "Cut the Rope". Although some features like cutting ropes needed to be added, the main functionality (physics, sprites, behaviors for collecting stars, etc.) was already available in the tool. The high-fidelity prototypes realized gave a clear *look and feel* of the games.

A tablet was chosen as target device because it can be easily ported to clients and end users and offers a more direct way of interaction, which favors its usability. Additionally, the system wraps low level physics simulation algorithms into high level elements such as ropes, squares, particle systems and other game elements like sprites, sounds, etc, which can be understood by anyone. These facts make it possible to create prototypes even during the first meeting between developer and client. This new prototyping technique pushes the benefits of prototyping to its limits. In traditional prototyping, the client first *explains* and in a later point in time developers *show*. The proposed approach allows requirements to be elicited in a single meeting through *collaboration* between client and developers.

Created prototypes can not only be used in the requirements elicitation phase. As opposed to other prototyping tools which produce throwaway prototypes, the prototypes produced by TangoPhysics are meant to evolve into the final application. The simulation engine integrated in TangoPhysics can be directly used in created applications. Furthermore, custom `PaletteItem`s can be reused across different projects. If a company specializes for example in software for intelligent houses, different elements like doorbell, televisions, telephones, wash machines, etc. can be defined, integrated to the palette and used in projects for different clients. Another example where reusability of elements may be of great advantage, is in games that have different levels (like "Cut the Rope"). In such games the same elements are repeated, but organized differently. What companies often do, is to create different scenario editors for their different games. TangoPhysics can be used as a multi-purpose scenario editor.

The current work offers many opportunities in the area of serious games. Integrating physical and particle simulations in a game represents a new and promising way of motivating players to learn physics, astronomy, mathematics, etc. The traditional approach where learners are presented with isolated tasks without a real challenge has failed in the market and fails to motivate players in schools. "Angry Birds" is a good example of a serious game with physics that is found to be extremely fun by many players. It is one of the most sold iOS games and it is still at the top of many iOS game ratings. TangoPhysics goes beyond a physics game; it is a physics game creator. Players can not only play the games created with it, they can also create their own games, which promotes learning. "It gives students a strong motivation to do a lot of research: they have to master the subject in order to be able to design a game about it" [6]. It could also be used by educators to create games that fit into their curriculum. Letting the educator create the game helps ensure "quality and relevance of the serious content featured in the game" [6]. The fact that public schools are typically limited in budget couples good with the *quick* prototyping character of TangoPhysics.

The system is composed of highly cohesive and lowly coupled subsystems, favoring its extensibility, one of its main design goals. If a company does not produce games with physics but shooting games, it only needs a different `World` Subsystem. If a game does not target the iOS platform but on Android, it only needs to add a `CodeGenerator` subclass to the `Code Generation` Subsystem. Finally, the `World` and `Simulation` Subsystems are designed to be used as-is in created applications. If both packages are copy-pasted in a new project, minimal code is needed to make the created application run outside TangoPhysics.

Future work is described next.

9.1 Supporting Polygons

One of the motivations of this master thesis was that defining polygons from the code is a tedious work. The current implementation supports only circles and squares. The reason why polygons were not included is because simple functionality for defining polygons would pose conflicts with usability requirements. The most simple way to support polygons is to let the user select its vertices. But users do not want to invest time defining vertices, specially when creating quick prototypes.

Ideally, an algorithm would receive an image as input and detect its collision polygon. This was indeed an objective of the project. But such a functionality could be a master's thesis on itself. Nevertheless, because this requirement was considered in the system's design, adding polygons' support should not pose major problems. The `PhysicalObject` is independent from its `Shape`. So adding polygon support requires basically adding a `Polygon` class to the `Shape` taxonomy. Moreover, `Polygon` class could have a constructor that receives an image as input parameter where the algorithm is performed.

9.2 Custom PaletteItems

The idea behind custom PaletteItems is reusability. The user can create complex objects made out of simple objects, and drag them into the palette. Once in the palette, the objects can be reused, not only in the same application, but across applications. Custom `PaletteItems` map to different classes when code is generated. So developers will be able to add behavior to custom objects. Other physics editors work with *plain* rather than *hierarchical* structures. They are therefore unable to define high level reusable objects to which behavior can be added.

9.3 Code Generation

Generating code was designed but not implemented. Objective-C code generation should not pose major problems. Functionality to simulate applications is completely independent from functionality to create and edit it. This functionality can be reused as-is in the game. Code also needs to be generated for custom `PaletteItems`, as already modeled in Object Design 6.4. Because CustomPaletteItems are built out of already existing PaletteItems, the code to generate custom palette items is minimal.

9.4 Snapping to Grid

Touching the screen to position objects is more intuitive than doing it with the mouse. But it is also less accurate because the finger occupies a considerable surface on the screen. The outer part of the fingertip looses contact before the inner part, and is often interpreted as a move gesture by the device and causes the object to move a few pixels.

Editors running on desktop computers typically let the objects be positioned with the cursor keys. It is also sometimes possible to control by how many pixels the objects are moved. For example, holding the control key makes objects move by one pixel. In this way, objects can be positioned very accurately.

In TangoPhysics, the `Scene` can be zoomed, allowing the user to perform any transformation more precisely. Nevertheless, snapping to grid functionality could help users save time in positioning objects. Snapping to grid is based on the idea that many objects in a graphical application need to be positioned on the same x or y coordinate. Snapping to grid makes an object that is being dragged, and that is only a few pixels away from a grid's edge, automatically change its position to match the edge.

9.5 Removing Objects

iOS uses the long press gesture to let the user remove the applications. When the long press is detected on any icon, every icon begins to tilt and displays an x on the top left corner that is used to delete the application. The same will be applied in TangoPhysics to remove objects from the scene. Using the same usability principles as in the iOS should make the users familiar with the application, enhancing its usability. This idea was indeed originated by a tester who could not find the remove button in the editing tab and tried with a long press gesture.

The fact that objects may overlap, hiding the delete button, poses a potential problem to this feature. To overcome this problem objects could enter a state in which if they are touched they just vanish. This way objects could be touched anywhere to be deleted, not only on the x icon. Furthermore, if two objects partially overlap, the scene can be zoomed making it easier to find a part of the object that does not overlap. Finally, if two objects completely overlap, one of them needs to be moved before they can be deleted.

9.6 Dynamically Loading Assets

Images, animations, sounds and music files used for the prototypical implementation have been statically included in TangoPhysics. Instead, assets will be loaded dynamically, from the device's Photo Album, Music Library, etc. In this way, not only the tool would accept any asset other than the hardcoded ones in the prototypical implementation. Images could even be taken with the device's camera, if available, and sounds recorded with the microphone. The quick prototyping character of TangoPhysics would be considerably enhanced. Instead of looking for images or sounds in the Internet, or investing even more time creating them with external tools and uploading them to the device, the images could be drawn in a sheet of paper, and directly uploaded to the device, and the sounds could directly be recorded with the device.

These features also widen the original scope of TangoPhysics. While the system can still be used for quick prototyping, many users will want to have fun with it creating small games. Indeed, the feeling of being part of the game is what many players like about them and the reason why many games let players choose among a very varied set of characters, for example. This system would not let the user select a single feature, but design an entire game. This was the idea behind Sketchnation, described in Section 4.1, which has succeeded in the market, although focusing on a very restricted kind of games.

9.7 Grouping Objects Around a Physical Object

Animations, Sounds, Triggers and Actions are currently permanently visible at the risk of overpopulated scenes. Because triggers, actions and animations are always associated to a physical object, the editing view of a physical object will offer a switch to hide and show these objects.

Grouping objects around a physical object will not only help reduce complexity of scenes. It will also help structure applications by concepts. For example, in a game where a ball vanishes using for this purpose an animation and a vanishing sound, every object around a ball (the action that makes it disappear, the trigger that triggers the action, the animation and the sound) will all be grouped around the ball, and will only be shown when the ball is selected.

9.8 Labels

Labels are often needed in different applications, to display scores or menu buttons in games, hints in other applications, etc. This is one of the few things the Rope Game prototype created with the system was missing.

`Label` objects will be added with the `font`, `size` and `text` properties. The `ChangeTextAction` will also be implemented, which will change a label's text when activated by a trigger.

Appendix A

Used Software

A.1 Software Development

XCode http://www.xcode.com/
IDE used for developing iOS applications.

Cocos2d http://www.cocos2d-iphone.org/
Framework on top of OpenGL for building 2D games, demos, and other graphical/interactive applications.

Box2d http://box2d.org/
2d physics engine written in C++. ilding 2D games, demos, and other graphical/interactive applications.

A.2 Software Resources

Gimp http://www.gimp.org/
Free open source image manipulation software.

Audacity http://audacity.sourceforge.net/
Free open source software for sound recording and edition.

A.3 Authoring this Document

MacTEX http://www.tug.org/
MacTEX is a complete TEX system for Mac OS X, supporting TEX, LATEX, AMSTEX, ConTEXt, XeTEX and many other packages.

LYX 2.0 http://www.lyx.org/
LYX is an open source document processor. LYX is a front-end to the LATEX typesetting system. It allows writing a document based on the structure of the document, not the appearance. This document is written entirely with LYX.

Visual Paradigm for UML http://www.visual-paradigm.com/
Visual Paradigm for UML CASE Tool supporting UML modeling with UML 2.1. All diagrams in this thesis are designed with this tool.

Inkscape http://inkscape.org/
An open source vector graphcs editor used to create the user interface presented in 4.7 on page 44.

Fooplot http://fooplot.com//
Function plotter with SVG support.

Appendix B

Glossary

Adapter Pattern A design pattern that let objects from unrelated packages collaborate by adapting one interface to another.

Artifact Any piece of software (i.e. models/descriptions) developed and used during software development and maintenance.

Boundary Objects Objects that represent the interface between the system and the actors.

Bridge Pattern A design pattern which decouples an interface from the implementations so that implementations can be substituted.

C A programming language described by the ANSI C standard.

C++ Object oriented programming language.

Composite Pattern A design pattern which represents a hierarchy of different objects.

Design Pattern Defined and repeatable design solution to a common problem. Design patterns are proven concepts for creating the code to solve a given problem. A common description of several design patterns can be found in [8].

Entity Objects Objects that encapsulate the application data.

Framework An extensible collection of classes providing a standardized interface to a particular service or application which can be then instantiated to build a concrete application.

Graphical User Interface (GUI) Visual interface of any application or tool. It serves as a bridge between the user and an application/tool and allows the user to input information into the application/tool in a graphic method.

Integrated Development Environment (IDE) Computer software that assists developers in developing software.

Memento Pattern A design pattern that provides the ability to restore an object to its previous state.

Object Design Document (ODD) Documents the object design by describing the decomposition of subsystems into packages, classes and class interfaces.

Observer Pattern A design pattern which maintains the consistency across the states of one publisher and many subscribers.

Protocol In objective-C, a protocol has the same functionality as a Java interface, or as a pure virtual class in C++.

Requirements Analysis Document (RAD) Documents the requirements elicitation and analysis by describing the requirements of the proposed system.

State Pattern A design pattern used to change an object's behaviour by changing its state.

Strategy Pattern A design pattern allowing objects to use different algorithms and change them at runtime.

System Design Document (SDD) Documents the system design by describing the design goals, subsystems, hardware/software mapping, persistent data management, access control, control flow and boundary conditions.

Unified Modeling Language (UML) An Object Management Group (OMG) standard for modelling software artifacts. It is widely-used industry-standard language for the specification, visualization, construction, and documentation of the software system components or software systems. It contains a set of symbols for creating diagrams to model software systems.

Testbed platform for experimentation of large development projects.

WYSIWYG What You See Is What You Get. Describes a way of editing data in which the data displayed during the edition process looks as in the finished product.

Bibliography

[1] Francesco Bellotti, Riccardo Berta, A. De Gloria, and Ludovica Primavera. A task annotation model for sandbox serious games. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 233–240. IEEE, 2009.

[2] Bernd Bruegge and Allen H. Dutoit. Object-Oriented Software Engineering Using UML, Patterns, and Java. August 2009.

[3] P J Cabrera, Joachim Bondo, Aaron Fothergill, Brian Greenstone, Olivier Hennessy, Mike Kasprzak, Mike Lee, Richard Zito, Matthew Aitken, Clayton Kane, and Oliver Hennessy. Starting with a Game Design Document: A Methodology for Success. In *iPhone Games Projects*, pages 129–152. Apress, 2009.

[4] Kevin Corti. Games-based Learning; a serious business application. *Informe de PixelLearning*, 34(6):1–20, 2006.

[5] Oliver Creighton, Martin Ott, and Bernd Bruegge. Software Cinema - Video-based Requirements Engineering. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06).* IEEE Computer Society Washington, DC, USA, 2006.

[6] Damien Djaouti, Julian Alvarez, and Jean-Pierre Jessel. Can Gaming 2.0 Help Design Serious Games? A Comparative Study. *Spore*, 1(212):11–18, 2010.

[7] Andr Filipe, Lessa Carregal, and Roberto Ierusalimschy. Tche - a visual Environment for the Lua language. In *In VIII Simposio Brasileiro de Computacao Grafica*, pages 227–232, 1995.

[8] Erich Gamma, Richard Helm, John Vlissides, and Ralph E Johnson. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 2000.

[9] Robert B Grady. *Practical software metrics for project management and process improvement.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[10] Brian Harvey. *Computer science Logo style (2nd ed.): volume 1: symbolic computing.* MIT Press, 1997.

[11] Stephanie Houde and Charles Hill. What do prototypes prototype. *Handbook of humancomputer interaction*, 2:367–381, 1997.

[12] R Ierusalimschy, L H De Figueiredo, and W Celes. Lua 5.0 Reference Manual, 2003.

[13] Roberto Ierusalimschy. Programming in Lua. *Library*, page 329, 2001.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 2–26, New York, NY, USA, 2007. ACM.

[15] Steffen Itterheim. iPhone and iPad cocos2d Game Development. *Development*, page 416, 2010.

[16] Chua C Kai, Leong K Fai, and Lim Chu-Sing. *Rapid Prototyping: Principles and Applications.* World Scientific Publishing Company, 2nd editio edition, 2003.

[17] Hoichang Kim, Taehyun Kim, Dongkyoo Shin, Dongil Shin, Soohan Kim, and Myungsu Lee. Design and implementation of intelligent physics system for mobile environment. In *2010 IEEE Symposium on Industrial Electronics and Applications (ISIEA)*, pages 329–333. IEEE, October 2010.

[18] John Kirriemuir and Angela Mcfarlane. Literature Review in Games and Learning, 2004.

[19] EMI Koivisto and Riku Suomela. Using prototypes in early pervasive game development. *SIGGRAPH symposium on Video games*, pages 149–156, 2007.

[20] Fabrice Kordon and Luqi. An Introduction to Rapid System Prototyping. *IEEE Trans. Softw. Eng.*, 28:817–821, 2002.

[21] J W Lloyd. *Practical Advantages of Declarative Programming*, pages 1–15. 1994.

[22] Gregory Maust. Algodoo in the Physics Classroom. *Education*, pages 3–5, 2009.

[23] Florian Mehm. Authoring Serious Games. *Educational Technology*, (June):271–273, 2010.

[24] MA: Merriam-Webster. *Merriam-Webster's collegiate dictionary.* Springfield, 11th edition, 2003.

[25] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Intersecting Classes and Prototypes. In *Proceedings of PSI-Conference.* Springer-Verlag, 2003.

[26] D R Michael and S L Chen. *Serious Games: Games That Educate, Train, and Inform*, volume October 31. Course Technology PTR, 2005.

[27] David Michael and Sande Chen. *Serious games: Games that educate, train, and inform.* Publisher: Muska and Lipman/Premier-Trade, 2006.

[28] Gerhard Zumbusch Michael Griebel Stephan Knapek. *Numerical Simulation in Molecular Dynamics.* Springer, 2007.

[29] Jakob Nielsen. *Usability Engineering.* Academic Press, Boston, 1993.

[30] Seymour Papert. Does Easy Do It? Children, Games, and Learning. *Game Developer*, 5(6):1–5, 1998.

[31] Susana Peciña, Barbara Cagniard, Kent C Berridge, J Wayne Aldridge, and Xiaoxi Zhuang. Hyperdopaminergic Mutant Mice Have Higher "Wanting" But Not "Liking" for Sweet Rewards. *The Journal of Neuroscience*, 2003.

[32] Marc Prensky. The motivation of gameplay: The real twenty-first century learning revolution. *On the Horizon*, 10(1):5–11, 2002.

[33] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, November 2009.

[34] Saul D. Rodriguez, Irene Cheng, and Anup Basu. Multimedia Games for Learning and Testing Physics. In *International Conference on Multimedia and Expo, 2007 IEEE*, pages 1838–1841, 2007.

[35] Katie Salen and Eric Zimmerman. *Rules of Play.* MIT Press, 2004.

[36] Jesse Schell. *The Art of Game Design.* Elsevier, 2008.

[37] Alexis Sepchat, Nicolas Monmarché, Mohamed Slimane, and Dominique Archambault. Semi Automatic Generator of Tactile Video Games for Visually Impaired Children. In Klaus Miesenberger, Joachim Klaus, Wolfgang L Zagler, and Arthur I Karshmer, editors, *Computers Helping People with Special Needs*, volume 4061, chapter 56, pages 372–379. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[38] S Siu, M De Simone, D Goswami, and A Singh. Design Patterns for Parallel Programming. *The 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 230–240, August 1996.

[39] A.J. Stapleton and P.C. Taylor. Physics and Playstation too: Learning physics with computer games. In *Australian Institute of Physics (AIP) conference. Retrieved July*, volume 18, page 2003, 2002.

[40] Andrew J Stapleton. Serious Games : Serious Opportunities. *Health Care*, pages 1–6, 2004.

[41] T Susi, M Johannesson, and P Backlund. Serious Games - An overview (x). *Skövde: University of Skövde (Technical Report HS-IKI-TR-07-001)*, 2007.

[42] Antero Taivalsaari. Classes vs. Prototypes - Some Philosophical and Historical Observations. In *Journal of Object-Oriented Programming*, pages 44–50. SpringerVerlag, 1996.

[43] Mary Ulicsak. Games in Education : Serious Games. Technical report, Futurelab, 2010.

[44] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, 2000.

[45] Electronic Version, Chris Crawford, Sue Peabody, The Art, World Wide Web, and Donna Loper. *The Art of Computer Game Design by Chris Crawford*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1984.

[46] Felix Willnecker, Damir Ismailović, and Dr. Wolfgang Maison. Architekturen mobiler Multi-Plattform Apps. In *Smart Mobile Apps*. Springer-Verlag, 2011.