A game commissioned by DOK and the TUDelft Library

BSc Project IN3405

Frank Baars (1262874)
Paul Bouchaut (1262904)
Miriam Doorn (1280333)
Nigel Karsidi (1257315)
Steven Sijbrand (1272950)

Delft University of Technology
EEMCS Faculty
23th of June, 2008

Exam Committee

Drs. E. Mantel
Delft University of Technology Library

Ir. M. Sepers
Dr.ir. A.R. Bidarra
Delft University of Technology
Preface

We worked on the development of 'Dark Ink' as part of our Bsc Education, namely the bachelor project. For this bachelorproject a varied range of projects was available to choose from. Because of our love for games however our eyes soon fell on the 'library game'-project. After talking to Liesbeth Mantel, one of the project leaders, and learning of the ambitious plans they had with the game, we soon came to the agreement that we would start our work april 7th.

Work on the project however started somewhat earlier. This game development project was started as part of the Ugame-Ulearn initiative. Ugame-Ulearn consists of three elements, a symposium on games and libraries, a game day at the library and a game development project with students from different educational backgrounds. Eight artists, two programmers and one project manager had already started working on the game in the beginning of the year.

We would like to thank the talented artists of our team from the Graphic Lyceum Rotterdam: Femke Vogelenzang de Jong, Frank Siebelink, Jason Teunissen, Jeffrey Vermeer, Marcus Jongste, Niels van Herk, Roeland van der Velde and Tom Uithol. Our fellow programmers from the Hague University: Niels Noppe and Taco Perquin. Our lovely managers: Inge van der Kruk (project manager) and Harald Warmelink (technical project manager). The project leaders, the ones that started it all: Liesbeth Mantel (Delft University of Technology), Jaap van der Geer (DOK) and Erik Boekesteijn (DOK). Remco Huijsser of Cannibal Game Studios for his valuable advice on implementation issues. Special thanks go to Arjan Egges, Ben van Basten and Folkert Konings of the Utrecht University, Department of Information and Computing Sciences, Faculty of Science for providing motion capturing facilities and expertise voluntarily. Also we would like to thank the 15 Beta testers who were patient enough to try out the Beta version and provide valuable feedback. And lastly we would like to thank our beloved Rafa for bringing this game development project to the MKT department.
Summary

UGame-ULearn was a project containing three different activities, a symposium about gaming in libraries, a game day at the library and a game developed by students. Wanting to create a game related to the library that is both entertaining as well as educational, UGame-ULearn started this game project. The game is called 'Dark Ink'. It's a hack & slash game with some educational elements. The educational part is represented by a minigame in which the user has to find information in a database, much as you would be looking for a book in a library or would perform a search on the internet.

Our assignment was to make a demo of 'Dark Ink' in collaboration with two students of the Hague University and eight students from the Graphic Lyceum Rotterdam. The demo should be playable and represent the game designed by the artist. The software design and implementation needs to be expandable so a new group of internship students can continue the work. Therefore the code is well commented and documented. We, as software developers, were responsible for the software design and implementation.

The design of the game is modular using different packages. First of all there is a ScreenManager which handles the different states the game can be in. It can handle various menus, but the most important screen is the GameplayScreen. Using the ScreenManager and separate screens it is easy to edit or add new screens. The game features some enemies, the so called minions, with advanced Artificial Intelligence. Using a flocking algorithm and simple planning method the minions can behave much like real creatures. We've researched and tested the flocking algorithm, also using Maple to balance out the various forces. The minigame uses a database that can be browsed in a special screen. Ingame you can pick up objects and start dialogs with so called riddlemasters. We were able to use the new Cannibal Composer, a tool that can be used by artists to import their content. Using this tool they could import all models and add the right textures, and create a level with all the models in the right place. There were some experiments with physics using JigLibX, but later on we removed most of JigLibX only keeping the parts used for collision. We weren't actually using a lot of physics and the physics engine was doing more wrong than good. We designed several gameplay elements. For user interaction we added a Head-Up Display (HUD) that would show the player's health. There also was an entire dialog system so the player could interact with various other characters. There is a camera system with different static cameras and sliding cameras. Both these different cameras and dialogs are activated using triggers. Basically the player's move method checks for intersection with certain trigger areas and if there is intersection it activates the right trigger.

This project had quite some challenges, both technical challenges as well as challenges in working with people from different educational backgrounds. In the end we are happy with the results, although we weren't able to do everything we wanted to do. Still, we believe the final product, a demo game, fulfills all expectations, especially seeing the little amount of time we had for this ambitious project.
# Contents

Preface................................................................................................................................. 2  
Summary.................................................................................................................................. 3  
Contents..................................................................................................................................... 4  
1 Introduction............................................................................................................................. 6  
2 Problem Definition and Analysis.......................................................................................... 7  
   2.1 Problem Definition........................................................................................................ 7  
   2.2 Involved Parties............................................................................................................ 7  
3 State Management.................................................................................................................... 8  
   3.1 Screens......................................................................................................................... 8  
   3.2 ScreenManager............................................................................................................ 8  
   3.3 Technical problems....................................................................................................... 8  
   3.4 Solutions...................................................................................................................... 9  
4 Artificial Intelligence.............................................................................................................. 10  
   4.1 Flocking....................................................................................................................... 10  
   4.2 Planning....................................................................................................................... 14  
5 The Mediabrowser.................................................................................................................... 20  
   5.1 Overview..................................................................................................................... 20  
   5.2 The InfoDatabase and Editor...................................................................................... 21  
   5.3 The PageInventory.................................................................................................... 22  
   5.4 The IngeScreen........................................................................................................... 22  
   5.5 The KeymasterScreen............................................................................................... 24  
6 Content Integration.................................................................................................................. 25  
   6.1 Game Object Composition......................................................................................... 25  
   6.2 The Cannibal Composer.............................................................................................. 27  
7 Physics & Collision.................................................................................................................. 29  
   7.1 JigLibX........................................................................................................................ 29  
   7.2 Collision detection using JigLib.................................................................................... 29  
   7.3 Computing Collision Primitives.................................................................................. 32  
   7.4 Issues using JigLib Collision....................................................................................... 34  
   7.5 Some simulated physics.............................................................................................. 35  
8 Gameplay................................................................................................................................... 36  
   8.1 HUD............................................................................................................................. 36  
   8.2 Dialogs........................................................................................................................ 37  
   8.3 Triggers........................................................................................................................ 42  
   8.4 Cameras........................................................................................................................ 45  
9 Conclusion................................................................................................................................ 47  
10 Evaluation.............................................................................................................................. 48  
Appendix A Research Report...................................................................................................... 49  
Introduction............................................................................................................................... 50  
Software Development Model.................................................................................................... 50  
eXtreme Programming.............................................................................................................. 50  
Conclusion................................................................................................................................... 51  
Prototyping................................................................................................................................. 52  
  Gameplay prototyping............................................................................................................ 52  
    The ‘Combo’ Gameplay prototype.................................................................................... 52  
    The ‘Dash’ Gameplay prototype..................................................................................... 52  
    The ‘Sword’ Gameplay prototype................................................................................... 52  
    The result......................................................................................................................... 53  
AI (flocking) prototyping............................................................................................................ 53  
Cannibal Composer..................................................................................................................... 53
1 Introduction

Delft has two big libraries. One of them is the DOK, a state-of-the-art library that doesn't just focus on books, but on all sorts of media. This includes movies, music, and also games. The other is the Delft University of Technology Library. It's the main library of the university, having a lot of scientific books. The two libraries had been working closely together, but never before on a big project, not until UGame-ULearn.

UGame-ULearn was a project consisting of three different activities. The most important one of them being an internship project where students of the Graphic Lyceum Rotterdam (the artists), and computer science students of Delft University of Technology and The Hague University (the programmers) worked together to create a game from scratch. The artists started first, making various concepts. After a lot of designing and discussing the final concept was chosen: Dark Ink.

Dark Ink is a hack-and-slash game. The world is at war with the Dark Ink. The Dark Ink is stealing all information out of books and the internet and attacking the humans. Armed with a sword the hero of the game, Tannic, goes out to kill all Dark Ink and retrieve the information they have stolen, while searching for a cure to his ink infection. In a more concrete sense this means that the player, as Tannic, navigates through a city and fights groups of minions, which are Dark Ink monsters. Next to the fighting, the game focuses on gathering information, just as you would need to do in a library. The minions sometimes drop information. You are helped by a holographic device, visualized by a little girl called INGE. She stores all data you find, and at certain points can gather intelligence by scanning a special building. In certain areas there are so called riddlemasters. This ink monster asks you a question. You then have to look for the right piece of data in INGE's database using the right search method to find an answer to the question. After showing the page that contains the answer, you receive color crystals. At the end of the level there is a keymaster. He holds the key you need to open the door behind him. To obtain that key you have to mix the colors of the crystals you've found to obtain the color wanted by the Key Master, much as in mixing colored paint. The game ends with a boss battle, where you have to defeat a huge Dark Ink Boss, defeating that Boss is the end of the demo that we have created.

In this report we'll start by explaining our assignment in detail in chapter 2: Problem Definition and Analysis. We then discuss the design and implementation on a per chapter basis. Because we used Agile Programming techniques there was no major design made at the start of the project. Instead a general design was made for the state and screen management and for the game environment. After this we started designing packages for certain features and then implementing them. These chapters are chapter 3 State Management, followed by chapter 4 Artificial Intelligence, 5 The MediaBrowser, 6 Content Integration and 7 Physics and Collision. Chapter 8 Gameplay will be about several smaller gameplay features. This chapter will be followed by a conclusion with recommendations. And then we close off with the evaluation of this project.
2 Problem Definition and Analysis

In this chapter we give the definition of the assignment given to us. Then we will give a list of all the people involved and their responsibilities.

2.1 Problem Definition

For this project, a working, playable demo of a game had to be made. The game itself must have elements that could make young people more familiar with old and new media and make them better at searching for and using information. The demo had to be playable for around 30 minutes. The target user group are children aged around 10 to 14 years old.

Products to be made

- APlayable Demo
- Technical documentation in order to enable better expendability
- Bachelorproject-report

Preconditions

- Users of the game should have a PC with Windows XP or Vista. They also need to have the .Net Framework 2.0 and the Microsoft XNA framework installed. In theory, with minor additions to the code, the game could also be played on the Xbox 360.
- The main language of the game will be English, so the users should be able to understand English.

2.2 Involved Parties

There are several parties directly involved with this project.

The development team consists of:

The Graphic Lyceum Rotterdam(GLR): A group consisting of 8 students of the GLR is involved with this project. Their team consists of three groups: 2 visualizers, 5 modellers and one storyboard artist. This group is lead by project manager Inge van der Kruk. Their responsibilities are the game design and the visualization of the design as in the textures, movies, models etc.

The Hague University (HH): The two students of the HH are responsible for the implementation of the game. They, together with the students of the TUD have produced all the code of the game. These two students also had the additional task of making prototypes before the actual coding of the demo began.

The Technical University of Delft (TUD): The TUD group consists of five Media and Knowledge Engineering students of the Software Engineering study. Like the students from the HH, they are responsible for the implementation of the demo.

The software development team (we) is responsible for reaching the final goals and conditions of this project. For most gameplay decisions we were also involved by giving feedback, insight and advice from a technical perspective.

One day a week, the technical manager Harald Warmelink discussed the state of the game with the programmers. His task is mainly steering the software development team into the right direction.

Furthermore, the project team of UGame-Ulearn is formed by: Erik Boekesteijn, Jaap van de Geer en Liesbeth Mantel. They are closely involved with this project.
3 State Management

The state management framework is used as the highest level of abstraction inside the main game loop. A game state is a purely abstract concept and is represented by a state of execution in which the application:

- listens to a certain set of inputs
- is able to output feedback to the user
- can transition to another state

These conditions are always true until the game is exited. This abstraction can be interpreted as a finite state machine. In this chapter we described how we handled states in our game by using Screens and a ScreenManager. Then some technical problems and our solutions are mentioned.

3.1 Screens

To simplify the implementation of the state management framework, we have chosen to represent a state by the less abstract concept of the interface which is used to give feedback to the user. We have called these interfaces Screens. In our implementation the relationship between states and screens is one-to-one, so this choice is only one of semantics.

3.2 ScreenManager

We have chosen to introduce a manager class which controls the behaviour of the individual screen objects. While each screen is, in theory, capable of handling any transitions to other screens as well its own deactivation, it is still beneficial to have a manager object which can take care of transitions and any other related tasks. Some of those benefits are:

- Centralized invocation of transition sequences
- Centralized memory management
- Better control of consistency (no duplicate instances for example)
- Ability to refer to a screen by id instead of object value or reference.

The last point listed means that the programmer can easily add or remove states/screens without breaking any existing reference which might exist because other screens are counting on the ability to transition to a particular other screen. This increases the programmer's ability to use agile design methods and iterative development practices.

3.3 Technical problems

One major difficulty with this design has been the loading of dynamic content (i.e. content which belongs to a certain game state but which can be different at arbitrary times of execution) such as level data which belongs to the "gameplay" state. The problem lies in the fact that we generally do not want to create multiple instances of a screen because each one belongs to a certain gamestate. It would be nonsensical to have duplicate states and it is arguable whether the definition of a new kind state is justified if only the content changes while the function stays the same. Imagine having a LevelA_State, LevelB_State, etc.

In other words, we wanted to reuse the instances of the Screen classes and just 'replace' the content of that screen. At this point we were presented with the problem that we could not modify any of the Screen members after instantiation, since the Screen instances manage themselves without much need for outside interference.
3.4 Solutions

The method we have applied to solve the content loading problems, is making the Screen classes which contain dynamic content accessible by means of several static methods. These methods can take a reference to the content location as an argument and internally handle the loading of the specific content type. We also added a Load and Unload method to the Screens that respectively adds all elements and purges all elements. This way a screen can be instantiated but empty.

Fig 3.1 Class diagram of screen management framework simplified for clarity
4 Artificial Intelligence

In general there are three types of artificial intelligence in the game. There are the small minions with limited planning abilities, and the bigger minion leaders with increased strategy. Both have extended flocking algorithms. The last is the boss also with increased strategy but for game-play reasons heavily limited in movement.

First in this chapter will be flocking and moving, followed by the planning of each of the three enemies.

4.1 Flocking

Forces are used to influence the movement of an NPC. Each enemy is affected by several forces that push or pull him to a certain direction. For example, between Minions there is a repelling Force, which causes Minions to move away from each other to a certain distance. Minions also have an attracting Force to Tannic, the main character who they want to attack.

In this section, state means a state in which a character can be, and not the kind of state the total game is in.

Design

Designing the flocking algorithm was started by describing different forces that would affect a minion. At first we started with very basic formulas for calculating the strength of a force and each type of force had its own class. Later on this was replaced with one Force class which would get the right parameters to calculate a force from the state the minion is in. This way different states would have different effects on the movement of the minions.

After some testing with prototypes we defined the following attracting forces:

- LeaderForce: attract minions towards their leader to form a group
- AttackForce: attract minions towards Tannic
- TaskForce: attract minions towards a certain strategical position

There also are several repelling forces:

- BuildingForce: prevent colliding with buildings
- CharacterForce: prevent colliding with other minions, leaders and Tannic.
A class structure for the enemies was designed too. The biggest difference between various enemy classes is the plan method which plans the future actions of the enemy. Other methods can usually either be used from the super class or extended in the subclass. Therefore a good design using inheritance was required.
Implementation

We have decided that a Force is very basic and that only the parameters for the Force calculation are different. These parameters are normally determined by the state in which a Minion or other NPC is. Therefore we only need one Force class. A Force can be created using different constructors, namely a non-argument class, one with float a, b and c and one with float a, b, c and boundary. The difference between the second and the third constructor lies in the idea that some forces are repelling and others are attracting. A repelling Force becomes less over distance and therefore has a lower limit of zero. Attracting Forces become larger over distance, and they have a possibility to explode. Therefore we provide an upper limit to this force, called the boundary.

Creating a new Force

Creating a new Force requires that in all the states the parameters a, b and c (and perhaps also the boundary in case it is an Attracting Force) are filled in. Often, these parameters differ from state to state, but it is not necessarily required. You are free to use the Force as a repelling or an attracting Force, but it is recommended that you use the guidelines described on this page.

Force Calculation

We have decided to give the Force class two methods, a calculateRepellingForce and a calculateAttractingForce method. Both take as arguments the position of the affected and the position of the effector. We also need a float measure as a function parameter, because we calculate a distance between the affected and the effector and then normalize it by the measure. More about the reason we added the measure factor can be found under the header Tuning of the Forces.

The Force will be calculated with the following algorithm, where a, b and c are all parameters provided by the state, the length is the factorized distance between the affected and the effector of the force:

\[
\text{Power: } (a) \times \text{length} ^ \text{(b)} + (c)
\]

In this formula (a) is general factor, (b) the type of exponential function and (c) the vertical displacement. You can see this in Tuning of the Forces.

This power is then clamped between zero and if specified an upper limit. The difference between an Attracting and Repelling Force is the direction of the Force. Upper limits are used for forces that get bigger when the distance increases, usually attracting forces. An upper limit is needed to prevent the forces of getting extremely big. If that happens it could pull a minion through a building. Repelling forces don't have to have an upper limit as these forces will only get extremely big when there is collision and if that case it's not a problem that other forces are neglected.

Add a Force

Forces can be added to each other using the AddForce method of the Force class. The input of this method is another Force. The easiest way to calculate a resulting Force, is by creating a new Force, that has zero direction and zero power. Next, we can add other Forces to this result, which in the end will result in an overall Force into one direction and with a certain power.

Tuning of the Forces

To determine the appropriate factors a, b, c and sometimes the boundary volume, we used the program Maple to calculate and draw the formulas with the given parameters. Below are some examples of figures we generated from the formulas. In this case, we used for length (which is the same as the measure parameter from the constructor of Force) to give a better impression at what distance there would be an equilibrium. As length, we used the dimension of a minion. With that, we were able to decide at what minimal distances Force should be strong and weak. This also makes forces independent of future scaling of the minion models as decreasing the
size of all minions (EG because they were too big compared to buildings), also decreases the forces. We use the boundary to give a maximum to a Force, to make sure an object would not overpower one force compared to others. If that isn't done, a Minion would ignore most forces and the idea of flocking is lost.

Fig 4.3 Forces in the CombatState

Fig 4.4 Forces in the Followstate

Fig 4.5 Forces in the IdleState

Fig 4.6 Forces in the TaskState

The inactive state does not have a special figure, because in the Inactive state all the forces are zero. We have also made some figures to make the measure MU (Minion-Unit) more understandable.

Fig 4.7 If the distance is 2, that Fig 4.8 If the distance is 1 then the Fig 4.9 As buildings have different means the real distance minions are colliding, at this point sizes, the size of the building between the minion is 1: only the CharacterForce gets should be subtracted so you have one minion can fit in between exponentially big roughly the distance to the wall
Moving

As stated above moving is done using flocking forces. The resulting force is used to calculate the new position.

First the algorithm checks if the power of the resulting force isn't too small. Because if it is, then it's rounded to zero. Small forces tend to switch direction very quickly which will cause the minion to change direction back and fro. Rounding these small forces to zero will just make him stand still. There's a special case when you are in the CombatState, in that case you want the minion to face Tannic and not to halt and stop moving when the force is too small. Therefore it does calculate a turning angle for this state, but if this angle is too small, then it is set to zero too, for the same reason.

If the resulting force is big enough then a capped speed is calculated. This is easily done by multiplying the power of the force with a speed factor and taking the minimum of that value compared with a default maximum speed. The minimum will make sure the capped speed will never be bigger than the maximum speed.

There also is a turning angle calculated. This is done by comparing the ForwardVector and the LeftVector with a direction Vector. The LeftVector is need to determine if the angle is positive or negative as we don't want angles bigger than 180 degrees (or Pi radians), else the minion won't take the shortest turn. The ForwardVector and the LeftVector are compared with the direction vector using the dot product. This is defined as \( u \cdot v = |u| |v| \cos \theta \). Therefore taking the arccos of the dot product (all vectors used in this method are normalized already), gives the angle in radians.

The size of the angle determines whether the minion should only move, turn slightly and move, or turn a lot and hardly move. The new position is calculated by adding the direction of the resulting force multiplied with the capped speed. This new position is forwarded to the general Move() method which also calculates if there is collision.

4.2 Planning

Planning is used for minions to set their state, or in case of the minion leader and the boss, also to set the state of all minions in their group. Using these various states the minions can have variable behavior and actually use some strategy.

Design

The design of the planning algorithm was done by making a basic flow chart and thinking about the different states needed. The goal was to make a plan method for the leader that was more intelligent than the one used by the minions. Basically the minion checks if he has a leader, if not he either idles or does a frontal attack. The leader has more intelligence as it can make his group patrol or send out some scouts. He can also order minions to defend or to attack.
**States**

Minions will have different states, each state determines how strong the different forces should be. This way you can make a state in which attack force is dominant, and a state in which attack force is completely ignored.

<table>
<thead>
<tr>
<th>Leader Force</th>
<th>Building Force</th>
<th>Task Force</th>
<th>Attack Force</th>
<th>Character Force</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inactive</strong></td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td><strong>Idle</strong></td>
<td>low</td>
<td>normal</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td><strong>Scout</strong></td>
<td>none</td>
<td>normal</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td><strong>Follow</strong></td>
<td>high</td>
<td>normal</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td><strong>Combat</strong></td>
<td>none</td>
<td>normal</td>
<td>none</td>
<td>high</td>
</tr>
</tbody>
</table>

*Minion hasn't been spawned/activated*  
*Minion has no task, is walking around a bit*  
*Minion is moving towards scout position*  
*Minion is in group attracted to leader*  
*Minion is attacking Tannic*

**Table 4.1 The strength of the forces in different states**

**Group Behavior**

Minions work together in a group. If one of the minions in the groups spots Tannic, he will set the TannicSpotted boolean in the Group class. This boolean, if set to true, starts a counter that resets the bool if Tannic isn't seen by one of the minions during a certain time. A group without a leader will go towards Tannic (minions get obtain his location by getting the character instance from the level). They are either in CombatState or in IdleState. A group with a leader will wait for the leader to give them orders. Normally the leader will first regroup all minions before attacking, instead of having them attack Tannic one by one.
Fig 4.10 A basic flow diagram of the leader decisions
Fig 4.11 Flow diagram of minion decisions
Fig 4.12 Flow Diagram for the Boss

Fig 4.13 Boss Arena, boss represented by square
Implementation

When the minion is active, a TimerEvent is fired after 0.1 seconds. This event fires a method called DoState(), which fires the DoState of the current State. This will be either take action, EG for the CombatState try to attack, or move. This is the same for the minion leader. Both the minion and the leader also have a second TimerEvent which is fired each second. This event fires the Plan() method, described in the flow charts above.

Minion Planning

If the minion doesn't have a leader, then he has to figure out what to do by himself. This will be either do nothing (without a leader we aren't that smart) or just attack if there is something to attack.

When he does have a leader he has to move towards his target position (for TaskState and CombatState) and when he is close enough he should call the DoState() method of that state. So for example is the minion is in TaskState and close to his target position, then he will look around and notify his group he has arrived at his target position. And if the minion would be too far away from his target position, he will call the move() method.

Leader Planning

A leader can access the positions of all minions in his group and Tannic's position. He can also see the health of his minions, of Tannic and of course his own health. These variables will determine the strategy he picks (with a certain random factor). When they haven't seen Tannic there are three possibilities: Send out some scouts to various locations, Patrol with the entire group between certain patrol points or idle.

When the group is fighting the leader can order the minions to use different strategies. He can put minions in defensive or offensive mode, while minions without a leader will always be in offensive mode. The basic tactic will always be to attack Tannic from the side and rear simultaneously. The leader will make his minions protect him, and more minions will protect him if his health is low. The leader will make anyone who is in front of Tannic go into defensive mode, and all others to attack. The leader will attack when he can hit Tannic in the back or when most of his minions have been killed. We can easily check the direction Tannic is facing using the ForwardVector, so it's not that hard to determine who defends and who attacks.

Boss Planning

A boss, on the contrary to leaders and minions, can't really move. But the boss has two special States. First of all it has a state for simple attacking, but it also have special state in which he can spawn minions that will be attracted towards him (using a FollowState) so he can absorb them and increase his strength and health. He will also have a special laser beam attack. This state will be set if at least one of the legs is heavily damaged. If he has absorbed all minions (or if the minions have been killed), he will return to the first state and resume the simple attacking.
5 The Mediabrowser

One of the requirements of the game was that it had an element of ‘mediawijsheid’ implemented. The term ‘mediawijsheid’, first coined by the Dutch Ministry of Culture, is about having a healthy mentality towards old and new media and its possibilities concerning information (www.mediawijsheid.org). The concept designed by the artists to implement this ‘mediawijsheid’ was a mediabrowser accessible in-game which holds information about objects, people, buildings etc. In this chapter we discuss the design and implementation of this mediabrowser. First an overview will be given of how the mediabrowser relates to the game and its story. The design and implementation of the mediabrowser mini-game can be split into two parts. The database will be described in one section, and the interface in another. The last section will discuss the second mini-game, the Key Master game that works with colors.

5.1 Overview

The game’s story tells how Dark Ink has stolen all information. In the game the player has a device called Inge that can record information. The information is represented in the game as a page in the database. Pages can be retrieved by defeating enemies, walking by certain buildings of interest or are received automatically after certain cutscenes and/or dialogs. You start out with a database carrying only a few pages and playing through the game this number would increase. In the game you would encounter Riddlemasters, who would give you a riddle, the answers of which could be found in the pages in the database.

In designing this part of the game a lot of corresponding with the artists was necessary. A lot of the details weren’t thought out fully yet and some of the ideas already there needed to be revised. For example, the question of how to answer the riddles; at first the idea the artists had, was to assign a code of symbols to the answers according to the position of the page in the database and the search symbols used to find. This however was a bad idea as the code would change when you added a page to the database upon finding one, and there would be different codes for each answer, as you could come upon them using different search symbols. Implementing this method would be troublesome and it also wouldn’t make much sense to the players of the game. In the end we solved this problem elegantly by adding a button to the interface of the database which would enable you to remember a certain page and use that as an answer to the riddle.
5.2 The InfoDatabase and Editor

The InfoDatabase is a class created to carry the information of the database pages, the questions (with answers) for the Riddlemasters and the associations of pages to buildings, dialogs and cutscenes. While I say database, we didn’t use an SQL database. We chose to have the information be read from and written to an XML file. We chose XML for its universality and ease of use. The XML file would be read only when starting a new game to fill the InfoDatabase object contained in the PageInventory, which will be discussed next. To make it easier for the storywriter of our project to edit the database, a simple editor was made using the standard Windows Application toolbox of Visual Studio C#.

As you can see in the picture the application isn’t exactly pretty nor is of high MMI standard. This wasn’t the goal though. The editor was required to function correctly, to supply the artists' needs, to be easy-to-use for our non-technically-educated artists, and to not be too time-consuming in its implementation.
5.3 The PageInventory

In the code the description of this class reads: ‘Class containing all things needed to handle the mini-games’, a very vague description but surprisingly accurate. With methods such as GetPagesForMonsters(int nr), which retrieves pages that can be dropped by the enemies when they die, and AllPagesFound(), which checks whether all pages in the area have been found, this class is a goto for all things related to the database and the mini-games.

Because the datapages dropped by the enemies had to be random and the questions asked by the riddlemaster as well, separate lists are in the PageInventory for the pages that have been found and the pages that can be found. When loading a new area, pages are assigned to buildings and monsters and consequently added to the pages that can be found list. Then, if there are riddlemasters in the area, questions are assigned to them. You have to be able to answer the question with a page that can be found in that area or a page you have already found in a previous area. Therefore the question given to the riddlemaster is chosen from a list of possible questions that gets questions added when the pages related to them are added to the pages that can be found list. There are a limited number of questions available (we had around 10), not every page has a question. You need to be certain that there is a question available for the riddlemaster to give. This is done by making sure that there are questions for the special buildings in the area that have a page attached. Then even if the pages that are randomly assigned to monsters have no questions related to them there is a question for the riddlemaster.

Fig 5.2 Class Diagram of the PageInventory

5.4 The IngeScreen

Each page in the database has certain categories or one category out of six possible categories
related to them. In the browser you can conduct a search on the pages you have found. You can select 1 to 3 categories to search on. When a page is found and added to the inventory it is also added to specific lists of the categories the page is related to. When searching on several categories the PageInventory calculates the intersection of the chosen categories' list. The lists are arranged in alphabetical order so the search is conducted very fast. A list of the search results is then given. Selecting a search result will show the information on that page.

The engine we were working with, the Cannibal Engine, didn't have many interface classes specified, they argued that interfaces are not that prevalent in games. Therefore a lot had to be implemented from scratch. For example: a scrollable search list, round buttons, buttons with text.

![Fig 5.3 Screenshot of the browser when selecting categories](image1)

![Fig 5.4 Remember Page Button](image2)

![Fig 5.5 Screenshot of the browser when checking search results](image3)
5.5 The KeymasterScreen

The ideas for a second mini-game was there for a long time. It wasn't until far into the development of Dark Ink however that they decided on what kind of a mini-game. In the end a mini-game with colors was chosen. When defeating a riddlemaster by answering his question correctly you receive color crystals. When you talk to the keymaster you are taken to the KeyMasterScreen where you can see the crystals you've already found arranged on the left. You can the drag and drop these crystals to the dark circle on the right. When pressing the surrounding button, the colors are mixed and you see the results in the bottom-right corner. The purpose of the game is to try and get the same color as the color initially in the bottom-right corner and around the mixed color, the so-called color locks. After successfully mixing colors a few times the keymaster is defeated and the way to the boss is cleared. The colors are mixed by taking the average of the RGB values of the colors. This method had the most natural results. The locks are calculated by mixing the colors of the crystals that can be found list in the PageInventory.

Fig 5.6 Screenshot of the KeyMasterScreen

Fig 5.7 Screenshot of the KeyMasterScreen after mixing colors
6 Content Integration

One of the greatest challenges in the making of games, is cooperating with people from a different "world": artists. Before a game is a game, a story has to be written and a game design made. Besides that, a game needs a set of data like models, with texture and animation; graphical interface elements; and level designs; all this data is referred to with the common term "content".

From the MKT4-project we brought the experience that when an artist has created a piece of content, there is a long way to go before it can take its place in the game. To name some of the work that has to be done: the content has to be exported from the art tool into the engine, and the appropriate code has to be linked to the piece of content. In our project we referred to this process as content integration. The common term in the game industry for the sequence of software elements that bring content into the game is content pipeline.

The sections below will describe the aspects of the content integration of Dark Ink. The first section describes the building of game objects from various elements. The second section describes the use of the Cannibal Composer.

6.1 Game Object Composition

To create game objects several different elements have to be combined. The graph below shows these elements and their relations.

**Mesh**

The mesh is the polyhedron form that defines the shape of the game object as it should appear in the 3D world. Meshes are created in programs like 3D studio max and were, in our project, exported to a file format called .fbx.
Material
A material defines what the surface of the mesh should look like. The mesh is, after all, only a shape; a material contains all the information needed to draw that shape onto the screen in a certain colour, lighting context etc.

Texture
The texture is an image drawn onto the mesh. It can give a character clothes and a face.

Shader
A shader is a piece of code that is responsible for drawing a model onto the screen. It defines the basic behaviour or "abilities" of the material. Typically this code is handled by the graphics device of the machine. It is extremely important that shader code is time-efficient because the shader draws each pixel of a model individually.

The shader handles texturing, lighting, transparency, and may even alter the shape of a model, for the purposes of animating the model for example. For details of all these, see below.

Material Parameters
A material parameter is a value that is passed to the shader but is constant for an entire model. The most common material parameters are lighting parameters, which define how much diffuse light the material reflects, how much ambient light it "emits", and its specular properties.

Opacity map
If the shader allows the usage of an opacity map, it can be used to make the model partially or completely transparent in some areas. This technique is essential to give a character "ragged" features such as hair.

Model
The mesh and material are combined into a Cannibal Model, which is then usable in the engine environment and can be attached to some Game Object.

Animations
To make a game object interesting it has to move, meaning not only changing position and rotation, but also truly animating the mesh.

Animations are added by the artist using some software, such as 3D studio max or MotionBuilder, into the .fbx file. The shader must process the animations for them to be played in the game.

Transformation
Models that are exported from artist tools are usually transformed in strange ways. Most commonly the models are sized a factor 100 to 1000 too large. It is also very common that a model is not centered in its own origin, while the origin of a character model should be between its feet. Therefore the model has to be transformed to be able to use it normally as a game object.

In the above graph this action is simplified. In fact transformations can only be made between two GameObjects, not between a model and a GameObject. Therefore to accomplish a correct transformation of the model two GameObjects are required, one that is assigned the model and a second which finally makes the connection between the first GameObject (using a transformation) and the code.

Code Class
Finally, a game object needs some behaviour. To accomplish this it has to be assigned a class of code that defines its role (and, consequently, behaviour) in the game.
Animation Code

To get animations to work, it is not enough to create them in a studio, save them into an .fbx file, and assign it to a game object. The code of the object must specifically call the right animation at the right time. Therefore animation coding is a specific step in the content integration process and can involve a considerable amount of work.

6.2 The Cannibal Composer

The content integration process of Dark Ink made great use of the pipeline software created by Microsoft XNA and Cannibal Game Studios. Simultaneous to our project, Cannibal has been developing the Cannibal Composer program, a piece of software that is designed to be usable by artists (who have no particular knowledge of coding or the Cannibal Engine), and that enables them to load their content into an engine environment to see how it functions there.

Dynamic object design using the Composer

As described in the section "Game Object Composition", a game object is created by assigning several elements to it, such as a model and a code class. All these assignments can be made in the Composer interface and stored directly into a content project file, a format used by Microsoft XNA to store and load content for a game.

The tricky step in the game object composition process, is defining its behaviour. Programmers can extend the Cannibal GameObject class to add special functionality to the specific Game Objects in the game. But objects of the GameObject class can also be saved into files using the Composer. To do this, one can create a GameObject in the Composer and assign a code class to each game object, so long as it is a subclass of the GameObject class. Consequently, when the object is loaded into the game from the content project, it is directly loaded as an instance of the given class.

It is this ability of the Composer to make the connection between model and code, that enables an artist to assign behaviour to their game objects, without actually knowing the details about code design, and test their game objects accordingly.

So the Composer allows all game object relations, as described in the first section, to be defined within the Composer interface. In theory, this allows an artist to load a character model as designed by them into the Composer, and run the game immediately to see how the model is doing within the game environment. If the artist is unsatisfied, he makes changes, loads them into the Composer, and runs the game again. All this can be done without a single line of code, which means that an artist can integrate his/her content into the game without help from a programmer.

In our project the content integration never got to this level. Artists would still have to be trained a bit for independently editing a Visual Studio project, and someone who knows how the engine works, how the Composer works, and how the game code works, would still have to supervise their activities. We thought the time expense of getting this type of content integration up and running smoothly too great for our project, which comprises only five characters and only a few other dynamic objects with associated content. For these reasons we assigned the task of getting dynamic objects to work with their content to a programmer.

Level design using the Composer

Contrary to dynamic game object composition, the composing of game areas was done by several artists, in the Cannibal Composer environment. This provided some insights into the usability of the Composer by artists.

The first step involved composing all individual static game objects such as buildings, lanterns etc. The second step was placing these objects into a larger GameObject that represents the terrain of a certain area in the game. The third step was placing dynamic objects onto the terrain.
In this way the Composer has proven particularly useful, effectively providing a "free" level editor.

**Issues using the Composer**

There were several concepts that proved too abstract for the artists. One of these was the complexity of the composition of a Game Object. The Composer does not make this process easier by lacking the feature of transforming a model when placing it into a game object. Transformations are only allowed between two game objects. To properly scale and position a model in a game object, a second game object has to be created, to which the first is assigned using the required transformation.

Also, because they were working directly in the game's content project, the artists had to make use of SVN, which is a tool specifically intended for programmers. Our suggested solution to this problem is integration a SVN client and a future version of the Composer.

Finally, the fact that the Composer is only in its beta stage caused some serious distress. The Dark Ink team has collaborated closely with Cannibal to solve the more critical problems, and created reports of any other problems encountered. In this way we have been a valuable beta testing group for Cannibal because of our intensive usage of the Composer.

---

Fig 6.2: Cannibal Composer: Level editor

Fig 6.3: Cannibal Composer: Character animation
7 Physics & Collision

Any game in which one moves a single character around a world requires, to some extent, game physics. Game physics are usually a simplified modelling of the physical rules of the real world, that allow objects to move around and apply forces to each other. Primarily they introduce gravity, collision detection and response, and some Newtonian behaviour of objects in the world.

7.1 JigLibX

At the recommendation of Cannibal Game Studios, we started using the free open source physics library JigLibX.

However, the effort required to get a game running with a physics system smoothly, was underestimated. We had hoped to put the physics system into our world and have it direct the physical behaviour of our objects without our having to think about it anymore, until of course we required some special behaviour of some object.

The opposite turned out to be true. A situation arose in which the physics system laid down the laws on us instead of the other way round. Objects flew around and bounced off each other and the ground, while the gravity pulled objects into a roll over the edge of the world. So we decided to re-evaluate our need for physics.

For our game we required the following:

- Collision detection & response (preventing characters walking through obstacles and each other)
- Walking smoothly across stairs

Remco Huijser from Cannibal advised that using a full physics system was only required when using very complex physics such as ragdoll physics (a character that is made up of multiple limbs and joints), or applying impulses to the edges of box-shaped objects to give them a rotation momentum. Simple physics involving Newtonian laws and collision detection could be done without such a heavy physics engine.

So for our purposes we decided to abandon the physics of JigLib and limit ourselves to collision detection. For this we used the collision detection algorithms and classes of JigLib, without waking up its physics system.

7.2 Collision detection using JigLib

To use the collision detection algorithm of JigLib, the world is assigned a CollisionSystem. We were lucky that the CollisionSystem was implemented separate from the physics system, so we didn't have to use this. Each object that should be an obstacle is then assigned a CollisionSkin. A CollisionSkin can be made up of one or more CollisionPrimitives, which can be a box or a sphere, or a complete mesh shape.

The next step is querying for collisions every time when an obstacle object is moved around. When a collision is detected with another object, the moving object has to moved back somewhat to its original position. However, if the algorithm moves an object back to its original position if it collides, a character could get stuck to a wall when he just brushes it. So another way of computing a "safe location" has to be found.

The solution is moving the character along the "sliding plane". This means that the movement vector of the character is separated into two components: one parallel to the surface it is colliding with and one perpendicular to it. The perpendicular component is then reduced to zero, so that the character no longer moves "into" the obstacle, but "along its side".

29
The process is more complicated when the character collides with two objects at the same time, for example when wedging itself into a corner.

For this we devised a quick solution, which is not completely mathematically correct, but accurate enough for our purposes. When a character collides with an obstacle, the movement component perpendicular to the collision surface is negated, producing a vector that moves the character back out of the obstacle, to a safe location. If multiple collisions occur, the outward vectors are simply summed up.
Below you will find the code of the collision response method.

```csharp
/// <summary>
/// Moves the object and checks for collision (absolute movement).
/// </summary>
/// <param name="newPosition"></param>
public virtual void MoveObject(Vector3 newPosition, Matrix newRotation) {
    Vector3 oldPosition = this.Position;
    Vector3 displacement = this.Position - newPosition;
    Vector3 direction = Vector3.Normalize(displacement);
    Vector3 safeLocation = this.Position;

    this.SetPosition(newPosition, newRotation);
    List<CollisionInfo> collisions = this.DetectCollisions();

    // the vector that should (after processing all collisions) solve all collisions
    Vector3 backoutTrans = Vector3.Zero;
    for (int i = 0; i < collisions.Count; i++)
    {
        CollisionInfo collision = collisions[i];
        for (int j = 0; j < collision.PointInfo.Count; j++)
        {
            // Get collision normal
            Vector3 N = collision.DirToBody0;
            float angle = Vector3.Dot(direction, N);
            if (angle <= 0)
                continue;

            // Compute safe location along sliding plane
            safeLocation = oldPosition + Vector3.Dot(displacement,
            // Compute translation to move from newPosition to safeLocation
            Vector3 backoutTransSingle = newPosition - safeLocation;
            backoutTrans += backoutTransSingle;
        }
    }
    // Go back to safe location
    this.Position -= backoutTrans;
}
```
7.3 Computing Collision Primitives

To assign collision primitives to our game objects we had to distinguish between dynamic objects and static objects.

Dynamic objects are characterized by their ability to move around, and particularly, to rotate. Rotation is a problem when assigning a collision primitive to a character. If the primitive is box shaped, for example, rotating may cause collisions where there were none before, without actually moving the character's position.

Therefore the logical choice for a character's collision volume is a sphere. The radius of the sphere could be computing using the character's model, however, this often produces strange spheres, particularly because the unanimated model is rather large because of stretched arms, or because the model is translated out of the origin when it is animated.

For static objects the logical choice seemed to be directly converting the model mesh into a collision mesh primitive. However, most of our models contained thousands of vertices, so for performance reasons we decided to approximate the collision primitives of static objects.

Most static objects in Dark Ink were buildings, and therefore mostly box shaped. Some exceptions were overcome by separating the models into different pieces, some of which were not designated as obstacles (such as little fences in front of buildings). The most efficient way to assign box-primitives to the objects, was writing an algorithm that computes them automatically.

The algorithm assumes the original model to be an axis-aligned box. Such a box is drawn that contains all the vertices of the model. After taking into account all transformations (such as the translation of an object that was not modelled in the origin), the primitive is assigned to the game object.

Below you will find the code that computes an oriented box primitive for a collidable object.

/// <summary>
/// Computes the dimensions and model offset of the world object.
/// </summary>
private void ComputeDimensions()
{
    // Retrieve model
    Model model = this.AttachedBody.Model;

    // Initialize intermediate result vectors
    Vector3 min = new Vector3(float.MaxValue, float.MaxValue, float.MaxValue);
    Vector3 max = new Vector3(float.MinValue, float.MinValue, float.MinValue);

    // Compute dimensions
    for (int i = 0; i < model.Subsets.Count; i++)
    {
        for (int j = 0; j < model.Subsets[i].VertexCount; j++)
        {
            // Get vertex semantic
            VertexSemantic semantic = model.Subsets[i].GetVertexSemantic(j);

            // Check semantic type
            if (semantic is PositionSemantic)
            {
                // Cast to position semantic if possible
                PositionSemantic vertex = semantic as PositionSemantic;

                // Compare with minimum and maximum
                if (vertex.Position.X < min.X)
                {
                    min.X = vertex.Position.X;
                }
            }
        }
    }
}
else if (vertex.Position.X > max.X)
{
    max.X = vertex.Position.X;
}

(etc, etc...)

this.dimensions = max - min;
this.modelOffset = 0.5f * (max + min);

// Apply LocalScaling of all nodes between this and AttachedBody, excluding
this.LocalScaling
Node n = this.AttachedBody.ParentNode;
while (!(n == this))
{
    this.dimensions *= n.LocalScaling;
    this.modelOffset *= n.LocalScaling;
    this.modelOffset = Vector3.Transform(this.modelOffset,
    n.RotationMatrix);

    n = n.ParentNode;
}

// Apply Translation of all nodes between this and AttachedBody to
modelOffset
this.modelOffset += this.AttachedBody.ParentNode.Position - this.Position;

/// <summary>
/// Computes the oriented bounding box for the Collidable
/// </summary>
/// <returns>Box primitive</returns>
public Box ComputeBoxPrimitive()
{
    // Compute bounding box position offset
    Vector3 offset = -this.Dimensions * 0.5f + this.ModelOffset;

    // Create collision box
    Box box = new Box(offset, Matrix.Identity, this.Dimensions);

    // Return result
    return box;```
7.4 Issues using JigLib Collision

JigLibX is obviously not a mature library yet.

Besides the fact that it was full of "TODO" comments (and few other useful comments), there were some obvious problems, for example, a CollisionSkin that has no CollisionPrimitives assigned to it, has a "world bounding box" that spans the entire world, crashing the program when querying for collisions.

A more specific problem was the placing of box volumes around buildings that were rotated. The rotation transformation of the object was applied to the volume, resulting in a box that was indeed rotated in the same way as the building itself. The position of the volume, however, was changed by applying this translation. In extreme cases the position of the volume could be off about ten times the size of the building. The problem only occurs when buildings are rotated in a non-right-angle manner, in other words, when the resulting box is non-axis-aligned. If the building is rotated a multiple of 90 degrees everything works perfectly. After several days of searching we could not explain this behaviour. Although we may have overlooked something we are inclined to say this is a bug in JigLib, in the best case a usability bug.

Fig 7.6 CollisionSkins made visible.  
Fig 7.7 CollisionSkins made visible.
7.5 Some simulated physics

Some effects that we wanted to implement in the game still required some small sense of physics, besides collision. At some points in the games Minions fly through the air, either because they are jumping towards Tannic or thrown away by Tannic.

This simple behaviour still does not justify a real physics engine, though, so we decided to "fake" the physics by simply moving the Minions in a parabola until they touch the ground again.

![Class Diagram](image)

Fig 7.8: class diagram of classes and methods playing some role in the Collision system
8 Gameplay

This part of the technical document is about the gameplay elements that have not yet been discussed and are smaller components of the game.

The components that are discussed here are:

8.1 HUD

Almost every game contains a Head-Up Display, a so called HUD. Also this game has a HUD. Our HUD is a Control, which means it can be placed on other controls (like the GameplayScreen) and also, controls can be placed upon the HUD. This makes it relatively easy to manage the HUD and its elements and to handle different resolutions.

The HUD contains the following elements:

- **AreaName**: Image that contains the Area name and will be shown for a short period;
- **ScreenText**: Text field that will show the picked up objects for a short period;
- **HealthBar**: Bar that represents the amount of health Tannic has;
- **ComboBar**: Bar that represents the amount of combo’s/adrenaline Tannic has;
- **ScreenFlash**: This will show up when Tannic takes heavy damage;
- **InkStains**: Object that creates an ink stain on the screen;
- **InteractionLabel**: Element that contains the message to interact with a person.

Indirectly, the DialogWindow is also placed on the HUD. This is done by the DialogWindow itself.

The HUD is kept simple and does not contain many difficult functions. Most elements of the HUD are shown and act using LineairInterpolationEvents, to make looking it smooth.

The methods that the HUD contains are not shown in the Class Diagram. Also some attributes are missing. Most of the missing attributes are events that are used by the methods of the HUD to show the changes in the HUD more smoothly.

Most of the methods in the HUD are used and called by other classes, outside of the HUD. When, for example, Tannic looses Health, he first performs his necessary methods and after that, the HUD is updated using the updateHealthBar method.

This is in a nutshell how the HUD works.

![Fig 8.1 Full Health bar and empty Combo bar. And almost full Health bar and slightly filled combo bar.](image1)

![Fig 8.2 Ink Stain.](image2)

![Fig 8.3 Screen Text.](image3)
8.2 Dialogs

Introduction

The Dialog system is based on the idea that a dialog is a connection of elements. When looking at a whole conversation, it can look like a big directed graph, that can be cyclic.

In this Dialog system, a Dialog can contain multiple Elements. These so called DialogElements are connected to each other. A DialogElement can have a pointer to the next DialogElement, sometimes to multiple DialogElements. One constraint is that only one next element can be loaded, so if multiple pointers exist in one DialogElement to other DialogElements, the one that is selected will be taken.

To start explaining in more detail, we begin at the bottom, the DialogElements.

DialogElements

Abstract Classes

A DialogElement itself is abstract. It contains some basic variables, like text, responder name and texture name (for the Image of the responder). Also, it contains two abstract methods, Perform and Close, that ought to be implemented. The first is called on the start, and lets a DialogElement show all his elements and set some parameters and events. The other one is called just before proceeding to the next element.
The second abstraction onto the DialogElement is the Selector. This abstract class is used for enabling the user to select a response to a next element. The selector therefore can have more than one next element, but according to the selected one, that one is taken. Cycling through the options is done by keyEvent’s that belong to this class.

*Implemented Normal Classes*

**Response**

The most common element that is used in a Dialog is the Response element. This element only contains the basic parameters of the abstract DialogElement class and simply sets the text. It is also possible to add certain tag’s into texts that will be removed by this element and replaced by GameVariables, like the movement and attack keys.

**NormalSelector**

This is the most common selector class. It contains all the basics of the selector class, and simply fills in all the options and the next elements that belong to these options.

**CloseDialog**

This element is the most special of all. It does not contain a text or an image. It simply handles that the Dialog should be closed, by calling the Close method of Dialog. This CloseDialog class was added for properly handling the closing of a Dialog, though it has not much more functionality.

*Implemented Special Classes*

**RiddleResponse**

This element demands that the owner of the Dialog is a RiddleMaster. More about ownership of a Dialog in the next part of this chapter. This element is different from the normal Response in the way that this element checks if an answer given by Tannic is right or wrong. If it is right, the RiddleMaster will disappear and a crystal is dropped. If the answer is wrong, minions will be spawned around the RiddleMaster.

**AnswerResponse**

This element is a selector, but it has some more advanced functionality. It is used for giving an answer to the riddle of the RiddleMaster. There can be multiple selected options which will lead to next elements. The option which contains the answer of the riddle only shows as an option if this answer (as a page) is remembered in the I.N.G.E. The last remembered page will be shown as an answer. From Gameplay point of view, the player should always have an option to refrain from giving the answer, when he is not 100% sure about the answer.

**Dialog**

The main class containing and handling a dialog is the Dialog Class. The relation between the DialogElements and the Dialog is as follows. The Dialog class contains a DialogWindow. This DialogWindow is used for all the visual effects and also has the list containing the elements. The reason why DialogWindow contains the elements, and not Dialog itself is because we wanted to evade a cyclic relation between Dialog, DialogWindow and DialogElements. DialogElements need to access the DialogWindow quick for placing labels and images. For this reason this structure was chosen.

The Dialog class has the methods to start and close a dialog, and contains the first element to be performed. This class also contains events that can be fired during the conversation. More about this later in this chapter.

A Dialog also has an owner, which is a Character. This gives the possibility to ask variables from the Character, like its position, and use it in the dialog. This also forces that a Dialog can only have
A Dialog also can be one of two types:

- Automatic: The dialog automatically goes from one element to the other;
- Manual: The player has to push a button to go to the next element.

**DialogWindow**

The Dialog window handles all the graphical parts of a dialog and it also contains all the DialogElements of a particular Dialog. A DialogWindow is a Control, which enables it to be placed on the screen, on top of other controls. In this case, we put the DialogWindow on the Hud. The graphical elements of a DialogElements, on their turn, are placed on the DialogWindow, which makes it easier to manage their screen positions and sizes.

The DialogWindow also contains several events. The AutoContinueTimer event and FinishEvent are events that handle going from one DialogElement to the next DialogElement. AutoContinueTimer event is a TimerEvent that fires every couple of seconds (class constant). The FinishEvent is fired when the player pushes the key on the keyboard to go to the next element.

Furthermore the DialogWindow class also fadeIn and –Out events. These are used for fading the DialogWindow and the elements on it in and out. These are mere graphical effects.

**DialogLoader**

In order to load dialogs from a file, the DialogLoader class was made. This class reads in an XML file, and returns the asked dialogs. It will not load all dialogs at once. Instead, it has a getDialog method that returns the dialog with the asked name.

The XML file should have the following format in order to be read properly (see Appendix B)

Unfortunately, due to the lack of time and the small amount of dialogs in the demo, there has not yet been written a user friendly Dialog Editor like there is for the I.N.G.E pages and questions. Therefore, we marked the Dialog Editor as a possible addition to the future.

**Special Events**

Some Dialogs will fire events. Other classes are able to subscribe their methods to these events.

The following special events currently exist:

- **CloseDialogEvent**: Event that is fired when a Dialog is closed.
- ExampleUser: The DialogTrigger (discussed in Chapter 8.3) subscribes to the DialogClosedEvent in order to show the red interaction Label again.
• **RiddleSolvedEvent**: Event that is fired when a Riddle is solved.  
  *ExampleUser*: The Gate class subscribes to this event in order to find out which riddles are solved.

**Possible Additions**

The Dialog system has the possibility to be extended in the future. Some possible extensions are:

- **Quest Dialogs**, with conditions and rewards. A possible way of implementing this, is by making a new Dialog and a Quest system, where the creator of the Quest can pick from predefined types of Quests and predefined types of Rewards. This enables that only the parameters of these Quests and Rewards need to be filled in.
- **Selector-option with precondition**. It is possible that you do not want to show all options yet in a Selector Element. Some options might have preconditions that need to be fulfilled before it will be available to choose. As a precondition, you can take other elements that already have shown, or that a certain elemental should not yet have passed.
- **A user friendly Dialog Editor**, that manages the Dialogs and make it easier to create them. It could be quite similar to the I.N.G.E. Editor for pages and questions.
Fig 8.8 Class diagram DialogElements

Fig 8.9 Class diagram Dialog
8.3 Triggers

In this section, the triggers that exist in the game are discussed. The Camera Triggers however, are discussed in the Camera section of the Gameplay Chapter.

Introduction

Triggers are important in almost every game. They often cause an action or event to be fired when some goal is reached. In this game, the Trigger class represents an area that has an event of action attached to it. Once the main character Tannic moves into the Trigger class, the preconditions are checked and the event or action is fired.

There are several extensions of the Trigger, each with its own implementation and special task. In this section, we will describe the DialogTrigger, the IngeTrigger, the MinionTrigger, the GateTrigger and the NextLevelTrigger.

Trigger

The main class for triggers in this game is the Trigger class. The Trigger class is abstract, and extends the abstract class Node. By doing so, the Trigger can now be placed into the world, using its position and rotation.

Furthermore, a Trigger has some additional parameters. The first is dimensions. The dimensions are used to determine the size of the trigger box, called the TriggerBody, around the trigger’s position. The trigger can also have a rotation, which can be set in the constructor.

The TriggerBody of the trigger can also be made Visible, though it was mainly used for debug purposes. It can also be used in the future for Level editing purposes.

The Trigger class has two main methods. These are:

- Intersect, which takes a point and calculates if that point is in the box. This method is basic to the class.
- Fire, which is an abstract class. Fire is called when Tannic intersects the TriggerBody of the trigger.

Firing a Trigger

In this short part contains an explanation of how a trigger is fired. The whole process starts with the main character Tannic. When he has moved, as in changed positions in the world, the method OnMove in the Tannic Class is called.

In this method, all the triggers in the world are queried using Tannic’s Bounding Volume. This means most triggers, that are too far away, are already filtered out.

The second step is checking if Tannic’s Position is in the Trigger Body. If so, the Fire method of that particular Trigger is executed. This is basically how the Trigger mechanism works.
One of the most used triggers in the game is the DialogTrigger. This Trigger is used for activating dialogs when Tannic enters the trigger. Before explaining the mechanism of a DialogTrigger, some main parameters will be summed up.

The parameters of this class are:

- Triggered Dialog. Trigger has a reference to a Dialog. This is the Dialog that can be initiated. A DialogTrigger can only have one Dialog.
- InitiateDialogEvent, which is a StoppingEvent<float>. This event will fire when the player stops pressing the spacebar. The spacebar is the key that enabled the player to continue through the dialog.
- Boolean isActivated that is used for checking whether or not a Dialog can be initiated.
- OutOfRangeCheck, which is a TimerEvent. This event will check every 0.1 second of Tannic is still intersecting with the box. If so, it can keep showing the interactionLabel from the Hud, and if not, the hideInteractionLabel method of the Hud is called.
- KeyboardDevice keyboard, which is a fast reference to a keyboard, for creating the stoppingEvent.

The methods of this class are as followed:

- The Fire Method of this DialogTrigger will do the following. It will call the Hud to show the interactionLabel, sets isActivated to true, and it will initiate both the outOfRangeCheck event and the initiateDialogEvent.
- The Deactivate Method will deactivate the trigger. It will remove the interactionLabel and
stops listening to the events.

- The initiateDialogEvent_Fired method will hide the interactionLabel, stops the interactionEvent and initiates the Dialog.
- The closeDialogEvent_Fired method will fire the trigger again when the Dialog has ended. In the constructor of the DialogTrigger, this method is subscribed to the CloseDialogEvent of the dialog of this trigger.
- The OutOfRangeCheck_Fired method will be called every 0.1 second and checks if Tannic is still intersecting with the bounding volume of the box. When it does not intersect with Tannic, the DeactivateMethod is called.

**IngeTrigger**

The IngeTrigger is used for letting Inge scan Special Buildings. SpecialBuildings contain information that can be stored as a Page in the MediaBrowser. When the main character enters the IngeTrigger, Inge will automatically pop up and starts scanning the building.

The IngeTrigger contains three variables. The first Variable is a pointer to the SpecialBuilding that should be scanned. The second variable is the Boolean isActivated, used for ensuring that the trigger will only fire once. The third variable is a pointer to the Level, for quicker access.

The Fire method will start with checking if it was not yet activated. If so, Inge will be taken from the Level, and placed on the main character’s position but a little bit above him. Then the Scan method of Inge is executed with the SpecialBuilding as its target.

**MinionTrigger**

The purpose of this Trigger is creating a so called booby-trap area that, once entered by the main character, spawns minions that jump over a wall to ambush the main character. This trigger will only fire once.

To the main parameters of this class belong the MinionWall, over which the minions jump. The second parameter is the Boolean isActivated, to ensure that the trigger only fires once. The last parameter is the number of minions that ought to be jumping over the wall.

The Fire method first checks if the trigger is not yet activated. If it is not yet activated, it will make a new EnemyGroup with the size defined in the constructor of this trigger. The minions will be spawned on the backside of the wall. Once spawned, the minions will jump over the wall using a parabolic function. Once on the ground again, they will go a new state which will then handle their behaviour.

**GateTrigger**

The Gate Trigger is used by the gate. It has several Dialogs that can be fired when nearing the Gate. The Gate has several objective parameters that need to be fulfilled in order to go to the next level. The GateTrigger acts as a medium between the Gate and the main character Tannic.

The Gate Trigger has several parameters. These are:

- Gate g. This is a reference to the Gate this trigger belongs to and it uses this Gate to get information about the objectives.
- Boolean hasFired. This Boolean is used for firing the Fire method only once when close. The Boolean will be reset once the outOfRangeTimer_Fired method is executed.
- The four Dialogs: MinionsLeft-, RiddlesLeft-, PagesLeft- and GateOpen-Dialog. Depending on the state of the objects, one of these is fired on entering the TriggerBody.
- OutOfRangeTimer, which is a TimerEvent. This event will fire every second and calls the OutOfRangeTimer_Fired method.

The Gate Trigger has two methods. The first method is the Fire method, which is the most important once of this trigger. In this method, the status of the Objectives is asked from the Gate
and depending on the current status, a Dialog is picked and initiated.

The `OutOfRangeTimer_Fired` method fires every second and checks if Tannic is still in the `TriggerBody`. If not, the `OutOfRangeTimer` event is disabled and the Trigger can fire again when Tannic moves again into the `TriggerBody`.

![Fig 8.11 GateTrigger around the Gate](image1)
![Fig 8.12 NextLevelTrigger around the lowered Gate](image2)

**NextLevelTrigger**

This Trigger is used for creating an area that, when entered by the player, initiates the process for unloading the current level and loading a new level.

The parameters are simple. The `String nextLevel` contains the name of the next level. This string is required by the method that loads the new level. It has a `Gate` to which it belongs, and it has a `Boolean` that makes sure that the trigger only fires once.

The `Fire` method does the following. It checks first if it has not yet already fired. If not, it will set its `fire Boolean` to true, and it calls the `DisableLevelEvents` of the `Level` attribute of `GameplayScreen`, and it calls the `NextLevel(String level)` method of `GameplayScreen` to start going to the next level.

**8.4 Cameras**

The game has several different types of cameras. We have a static camera that doesn't move, and two types of FollowCameras that either follows the character from the rear or from the side. These FollowCameras only move into one direction and are not attached to the character. There's also a FreeCam, which is used for testing and debugging purposes only.

**Design**

Designing the cameras was pretty hard. First of all the artists didn't exactly know what type of camera system they needed, and their type of camera changed almost every week. Secondly, if they did think of a system, then it was a rather complex camera. To solve this we have successfully simplified the complex system to a system we could implement within the deadline. This design and the prototype was then approved by the artists and built into the game.
Implementation

As stated above, there are multiple cameras. First we'll explain how the camera triggers work and how the camera changes position. After that we will explain how the independent cameras work.

Triggers

In the level there are various triggers. Each type of camera has its own trigger. The trigger will create a DarkInkCamera object that has a reference to the Cannibal Camera and a method to update the position of that Cannibal camera. When a new DarkInkCamera is set in the level (called CameraHolder, as it is not a real camera), it deactivates the old DarkInkCamera and activates the new one. Activating and deactivating is only done to start and stop the method that periodically updates the position of the Cannibal camera. This way we can simulate different types of camera while only using one real camera.

Camera

There is a static camera, two FollowCameras, one for rear following and one for the side, and a free camera.

The FollowCameras have a start and an end position. To make it a bit easier the camera will only move parallel to either the X-axis or the Z-axis. The direction the camera has to move is obtained by calculating the smallest difference between X and Z of the start and end. By multiplying this direction with the position of the target we get a position independent of moving into the other direction (the one the camera doesn't move to). After that we only have to calculate the corresponding Y-coordinate and then clamp all values between the start and end position. Last we call the LookAt method which also turns the camera when the character would walk outside the area between the start and end position of the FollowCamera.

The free camera can only be moved using keyboard controls. There are controls for moving forward and backward, turning left and right, and panning up and down. The freecamera is available by typing a secret phrase while playing the game (cheat code).
**9 Conclusion**

The nature of the project - making a computer game with mainly students with little game development experience - has put us in a special position. We have had the opportunity to play an integral part in the development process and gain a lot of first-hand experience. Not only did we learn much on a technical level during the months we have worked on this project, but we have also got more skilled at teamwork and creative thinking in the broad sense of the word as well as becoming more aware of management challenges.

**Challenges**

Computer games often consist of many different elements which all may require different skills and knowledge to create. This was also true in our case, we have created frameworks for artificial intelligence, state management, level loading, user interfaces and game behaviour frameworks which govern whatever might happen in the game, just to name a few. Creating each of those elements, and many others, presented new problems. While creating these individual building blocks posed quite a challenge already, integrating these into a complete computer game was at least as challenging.

Besides the technical challenge, we have also experienced the necessity of good communication with other programmers as well as non-programmer team members. The creative nature of the game development process forces team members with different specializations to understand each other's way of reasoning and have at least some understanding of the limitations which are imposed on people with other specializations.

**Reflection**

The final product which we have produced is largely completed and most planned features were implemented, so we are glad with the result of several months of work. That said, we did not get around to doing everything we set out to do. While we never assumed we would be able to do everything we wanted to do, some improvements could be made with the knowledge gained. One of the hardest things to get right was probably the time management, which goes for the programmers specifically as well as for the project as a whole.

With the insight we have gained over the past months, we believe we could probably allocate time and man-power more efficiently, since it is easier to estimate the resources required to complete a certain task now we know how much time certain tasks actually take. An example is the beta testing phase, which we had scheduled in the final two weeks before final release. The intentions were good, but releasing and distributing beta took too much time for such a testing phase to be as effective as it could have been.

**Final words**

In conclusion, we considered creating this computer game as quite an ambitious project. We predicted many challenges beforehand, which we found to be an accurate assessment in retrospect. Besides experiencing some of the technical challenges that come with such a project, we also have had the opportunity to experience, and learn from, working with a relatively large interdisciplinary team under the pressure of the money and resources invested by the project managers.
10 Evaluation

This BSc-project is the largest project of our Bachelor Curriculum. All time in the last half of the sixth semester was used for this project, with the exception of some hours for the seminar course. Sometimes this was a bit annoying, as you are focused on the project, but then have to work for the seminar. It would be better if this seminar would be about the same topic as the BSc-project, although there still was enough time left for the project.

We were very happy working for UGame-ULearn. It was a great experience working with a lot of other internship students. Especially working with artists is always a good practice. The downside of having a project with internship students is that you can't actually learn anything from the professionals. If you were working at a real game studio, there would be a professional who could give you advise and feedback, which could be very enlightening. The location of the workplaces were alright too, having a private room at the university campus. Unfortunately the workplaces were a bit less ergonomic.

The assignment, creating a computer game, was very exciting. It was a big project where everything had to be made from scratch. This gave us a lot of freedom to use our own ideas and designs. We chose this project because we were excited about the assignment, it proved to be a challenge.

The teamwork was great. Being in the same room all of the time certainly helped. The communication between the software developers, with a few exceptions, was very well. Also the communication with the artist team was excellent. Unlike with the MKT4 project, with hardly any direct communication, we could quickly discuss designs and test prototypes. This way we, as software developers, were hardly delayed by the artists and game designers.

The BSc-project was very good practice to end the bachelor curriculum. We certainly needed all knowledge of all other classes and experience of the previous projects. Working with eXtreme Programming is very difficult, it really requires motivation and organization of each of the individual team members.
Appendix A Research Report

Research Report

BSc Project IN3405
Frank Baars 1262874
Paul Bouchaut 1262904
Miriam Doorn 1280333
Nigel Karsidi 1257315
Steven Sijbrand 1272950

Delft University of Technology
EEMCS Faculty
23th of June, 2008

Exam Committee

Drs. E. Mantel
Delft University of Technology Library
Ir. M. Sepers
Dr.ir. A.R. Bidarra
Delft University of Technology
Introduction

In this research we have investigated certain things needed for our project. We have looked for and adopted, what we think, is the best software development model. We also made various concepts of parts of the game, called prototypes. We will also shortly discuss the Cannibal Composer, a useful tool for content integration. The last chapter will describe the flocking algorithm and all research done for that algorithm.

Software Development Model

Making a design for a computer game is incredibly hard. There are multiple reasons for this. First of all you are dependent on artists who make up the story for the game and most game-play decisions. The story is never finished when you are starting your design. Actually, it is even worse, the story is not just not finished, and it will most likely change. Also certain gameplay elements can only be tested using play-testing. This means that certain elements of the game are likely to change after they have been designed and implemented. Games usually use the latest technology too, this also includes technology that was not available yet when the game was designed. All these things together require a very well software development technique.

eXtreme Programming

eXtreme Programming (XP) is a software development technique linked to Agile Software Development. The biggest difference of using an Agile method above the classic waterfall method, is that you use iterations and therefore are able to change the design. To overcome the problems that the classical waterfall method would have when creating games a number of well-known software developers made a few principles for Agile Software Development, called the Agile Manifesto. A few key principles are simplicity, self-organizing teams, many iterations, adaptive and teamwork using direct contact.

Using the Agile Manifesto eXtreme programming defines a few best practices. These practices have been used before, just not all combined. Some best practices used by eXtreme Programming are:

Simple designs, ongoing refactoring and integration, developers work in pairs, have the same rights and use the same coding standards and use user stories (spikes).

Simple designs are needed if you want to be able to change the design later on. If a design is too complex it will be nearly impossible to change it. However, if you use simple and modular designs you can easily modify a part. It is also important to keep the code as simple as possible. Note that simple does not mean as less code as possible, but working and well structured code without any extras. To obtain this it is also need to keep refactoring all code. Refactoring simplifies the code which is also very useful when you are going to integrate other code.

Developers should work in pairs, first of all two people know more than one, and thus they can think of solutions for problems more quickly. Also the second person can check the code right away and detect bugs while the code is still being written. Because all Agile methods use less documentation, communication is more important. Therefore all developers have the same rights so everyone can refactor each others code. This does require a lot of direct contact and the use of the same coding standards.

Last but not least, eXtreme programming uses user stories, also called spikes. A spike usually represents one feature. By using spikes multiple features can be implemented at the same time by

---


different teams. This also improves the modular design. The spikes however have to be integrated into the rest of the software, which brings us back to the first point: simplicity. Without simple designs and ongoing refactoring it will not be possible to keep integrating new parts.

**Conclusion**

We believe that using eXtreme Programming may solve the design problems we will have when developing a game. A simple and adaptive design will result in simple and adaptive code. This will be needed as code will change, some things just cannot be tested until they are fully implemented. eXtreme Programming however asks a lot of communication and teamwork. This is a challenge, but in our opinion we did succeed.

Also, as we are developing a demo that may be extended by other developers, this development method will assure that the new developing team will be able to continue with our design and code. If it is simple and adaptive, they can easily understand and extend the current design and code.
Prototyping

Introduction

Because of the nature of Game Design, which is a creative process, a Game Design decision cannot be made until the alternatives have been thoroughly researched, tested and evaluated. That means that several alternatives have to be worked out and implemented, before a definitive choice can be made. For the purpose of testing these alternatives simple games have to be made, that are not supposed to be full games, of high graphical quality or code maintainability, but are instead quick tryouts that are meant to be tested once and then discarded. These simple games are called prototypes.

In Dark Ink a number of game design decisions had to be made using prototypes. An example of such a design decision is the combat Gameplay. The problem is discussed in more detail below. Some alternatives are discussed and finally the result of the evaluation of the prototypes is given.

There were also some other aspects of the game, which didn’t necessarily involve decision-making, that had to be tried out before they could be implemented in the common project. An example of such an aspect is the flocking algorithm.

Gameplay prototyping

The game design had to involve some form of entertainment, besides the educational part. The general idea was that Tannic, the main character, would be attacked by groups of Minions, Dark Ink monsters, which he should then fight and defeat. How exactly the fighting would be done wasn’t decided yet. Several ideas came up and they had to be prototyped before the decision was made.

The ‘Combo’ Gameplay prototype

One alternative was using the widespread combo idea. Like in many console games, our character would have 2-4 simple attacking abilities, accessed by buttons on the controller. However, the fight can be made more interesting by using special combinations of those buttons, which makes the character make aggressive movements that are more powerful than the sum of the simple movements associated with the button combination.

Another common element in games with such Gameplay, is the fact that the character’s strength or abilities grow as he destroys more enemies at a time.

For the demo of Dark Ink, we chose a very simple implementation of the combo system, involving only two buttons and some very simple combination mechanism. The system will not be described in full detail here. Suffice to say that it focuses on a rapid Gameplay with a very fast Tannic.

The ‘Dash’ Gameplay prototype

The second alternative involved Tannic being much quicker than the sluggish Minions. Tannic would have to dash around and find a weak spot of an enemy, for example the back, and stab it once with a lethal blow. To be able to do so Tannic would have the ability to dash, make a short, quick jump in some direction. The player would have to press the attack button at just the right time to hit a Minion in the crucial spot.

The ‘Sword’ Gameplay prototype

The last Gameplay alternative involved a more intelligent Gameplay. The player would be able to control in what way Tannic strikes his sword, to be more specific, in what direction he would strike. A fight with a Minion would resemble a game of fencing, in which the Minion can block several
attacks from Tannic, eventually exposing a weak spot which the player should exploit. This Gameplay would involve less Minions because Tannic wouldn’t be as quick in defeating them as in the other two Gameplay alternatives.

**The result**

All three Gameplay alternatives were prototyped, tested and evaluated. The results were that the combo prototype best suited the requirement of fast-paced action and rapid Gameplay. To make a long story short, the sword prototype was not immersive enough, and the dash prototype was too difficult in the sense that it would be more frustrating for the player to get to the weak spots of all the enemies than it would be exciting.

**AI (flocking) prototyping**

The artificial intelligence of the enemies was decided to involve a flocking algorithm. After designing this algorithm it had to be prototyped before it was implemented in the game itself in full. A flocking algorithm must be subjected to a lot of ‘tweaking’ to function optimally; the parameters of all forces in all circumstances have to be determined. For these reasons a prototype was created which demonstrates the behaviour of the enemies.

The flocking algorithm is described in more detail in the section flocking of this document.

**Cannibal Composer**

When an artist has created a piece of content, there is a long way to go before it can take its place in the game. To name some of the work that has to be done: the content has to be exported from the art tool into the engine, and the appropriate code has to be linked to the piece of content. Fortunately all these activities do not have to be done in code; tools exist for these purposes, and the tool we used is the Cannibal Composer.

The composer creates files of the content project type, which is a file type, or actually a folder structure, used in Visual Studio to describe and load all content into the game. This enables one to use the composer to load content directly into the visual studio project.

The interface of the composer allows one to create objects of all classes used in the Cannibal Engine, and also subclasses of those. This makes it possible to design and create entire areas and ‘levels’ in a game without touching a line of code.
Flocking Algorithm

Moving Artificial Intelligence
When we were asked to create artificial intelligence for the game, we started looking for various algorithms. The two most common methods are path-planning and flocking. The advantage of flocking is that you don’t have to calculate paths which can be very inefficient, whereas path planning may require a lot of time. Flocking is also very variable and can easily be tweaked. Using path planning it would be more difficult to take other AI into account, but with flocking it’s just adding an other force. We also had previous experience using flocking.

Flocking Algorithm
The basic outline for the algorithm is first to calculate all separate forces, add these forces up and then use the resulting force to make the final move.

Calculating a force is done by querying all objects, then for each object that inflicts a force the power and direction is calculated and then added to the resulting force. The power can depend on the distance between the object and the AI, or it can be fixed. The direction could be towards the object or pointing away from the object. This should be defined per type of force. The standard types are repelling force between objects, attracting force between AI members and attracting force to target.

The forces are added using a standard add method for 3D-vectors. This method can handle different directions with different powers and calculates a resulting force.

Finally, movement of an object is done by a translation into the direction of the resulting force with a distance proportional to the power of the force.

Conclusion
The method suggested is a good algorithm to start with. It is easy to implement and can easily be tweaked.

Avoiding Objects
Another part of the flocking algorithm is to avoid static objects. Preferably this is done without doing real collision detection as it looks rather fake when an AI object first walk into a wall and then slides around it. It would be better if the AI would neatly walk around the wall without hitting it. There are multiple ways to do this. First of all it can be done using forces that would push minions away from static objects. A second method would be checking for collision at a point in front of the AI.

Algorithms
The first algorithm would be an extension of the basic flocking algorithm. The biggest disadvantage of this algorithm is that there are different object sizes. Just calculating the dimensions will not always be sufficient. Using this method you will assume all objects are spheres. You could however, improve the method a lot by calculating the distance and then subtracting the distance to the plane of the box. This last distance can be calculated using an Axis Aligned Box.

The second algorithm requires a method to check if a certain point is inside a building. There are several ways to solve this. All methods here are taken from “Steering Behaviours For Autonomous Characters” written by Craig Reynolds.

Object Avoidance
The first method, called Obstacle Avoidance, checks for collision using a rectangle in front of the

\(^{3}\)Craig Reynolds, “Steering Behaviors For Autonomous Characters”. Last checked: 19-06-2008
Website: [http://www.red3d.com/cwr/steer/](http://www.red3d.com/cwr/)

54
AI. The size of this rectangle is dependent of the velocity of the AI. If there is collision, the nearest collision object is taken, and a steering force is added, which is opposite of the direction to the centre of collision object and perpendicular to the moving direction. Also a braking force is added.

![Figure 1 A schematic overview, magenta is the velocity, blue is the steering force. Image from: Craig Reynolds, http://www.red3d.com/cwr/steer/Obstacle.html](image)

**Containment**

The second method, Containment, uses three probe points in front of the AI. One point is in the same direction as the velocity, two others are somewhat diagonal. When one of the points collides with an object, then that point is projected to the nearest point on the surface of the object. Then a steering force is giving to the AI in the direction of the component, perpendicular to the velocity of the AI, of the surface normal at the previously projected point.

![Figure 2 A schematic overview: white lines are the probe points, red is the surface normal and blue is the steering force. Image from: Craig Reynolds, http://www.red3d.com/cwr/steer/Containment.html](image)

**Conclusion**

All algorithms are very powerful. However the last two: Obstacle Avoidance and Containment, require a velocity, which our system doesn’t use. Although it is possible to introduce this, using Newton’s first law, by adding a force proportional to the current speed of the AI. The biggest disadvantage of this velocity, is that, as in Newton’s first law, it takes a force for an amount of time to slow down the AI. This would work perfectly for vehicles, but for creatures that run through narrow streets, we decided not to use velocity. That left us with a problem, without velocity these methods just don’t work. The steering force should push the AI away for a while, and not just the tiniest moment when there is collision.

The other method however, using a repelling force, was very simple and worked reasonably nice. Though this method could be improved not by only taking the smallest distance to the object,
but also using a force in the opposite direction of the surface normal of the closest point on the object, much like in *Obstacle Avoidance*. 
Appendix B Dialog XML File

<?xml version="1.0" encoding="utf-8"?>
<data>
<dialog>
<dialogName>area_1</dialogName>
<firstElement>1_1</firstElement>
<dialogType>Automatic</dialogType>
<elements>
<element>
<type>response</type>
{name>1_1</name>
<responder>Leader</responder>
<image>Leader</image>
<text>How is it possible that a pathetic human made his way through?</text>
<nextElement>1_2</nextElement>
</element>
<element>
<type>response</type>
{name>1_2</name>
<responder>Tannic</responder>
<image>Tannic</image>
<text>Just die by my sword and get this over with!</text>
<nextElement>1_3</nextElement>
</element>
<element>
<type>response</type>
{name>1_3</name>
<responder>Leader</responder>
<image>Leader</image>
<text>Hahaha, my minions will deal with you!</text>
<nextElement>1_4</nextElement>
</element>
<element>
<type>close</type>
{name>1_4</name>
</element>
</elements>
</dialog>
<dialog>
<dialogName>riddlemaster_3</dialogName>
<firstElement>RiddleDialog_One</firstElement>
<dialogType>Manual</dialogType>
<elements>
<element>
<type>response</type>
{name>RiddleDialog_One</name>
<responder>RiddleMaster</responder>
<image>RiddleMaster</image>
<text>...</text>
<nextElement>RiddleDialog_OnePointOne</nextElement>
</element>
<element>
<type>response</type>
{name>RiddleDialog_OnePointOne</name>
<responder>Tannic</responder>
<image>Tannic</image>
<text>Just give me a riddle.</text>
<nextElement>RiddleDialog_OnePointTwo</nextElement>
</element>
</dialog>
This is my riddle: {RIDDLE}

Choose a response:

- My answer is
- Hmm, I don't know the answer yet. Maybe I should check INGE...

The answer you gave is