Systementwicklungsprojekt:
BreakOut - Augmented Reality for computer games

Steven Pessall
pessall@in.tum.de

March 29, 2004

1 Introduction

This document is a result of my "Systementwicklungsprojekt" (SEP) conducted at the chair for applied software engineering at the Technical University of Munich. This chapter deals with the motivation for this project. Additionally there is a short overview over the implemented game and over related research projects.

1.1 Motivation

Demonstrating the possibilities of augmented reality systems to an audience lacking in-depth knowledge can be quite difficult. The aim of this project was to implement a simple game which can be used to demonstrate the possibilities of augmented reality in general and particularly of the DWARF framework.

1.2 Gameplay

The game chosen for this purpose is based on the popular game BreakOut, which was originally developed by Atari in 1976 as an arcade video game. Over the years there have been numerous remakes and similar games.

In this game the player controls a racket at the bottom of the playing area. The player has to use the racket to prevent the ball moving in the area from reaching the bottom side. At the top of the area there is a number of blocks, which have to be destroyed to win the game. The blocks are destroyed if they are hit repeatedly by the ball.

In the further course of this work the term "BreakOut" refers to the game developed during this SEP project.

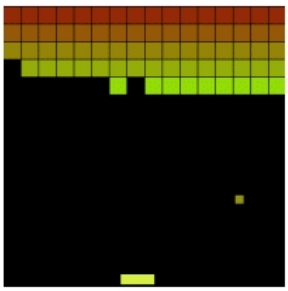

Figure 1: BreakOut

1.3 Related Work

There have already been several projects with the aim of implementing games in AR for the purpose of demonstration. Like BreakOut these are very often based on well known existing games, but differ greatly in the degree in which they make use of augmented reality aspects. The list in figure 2 is a selection of projects and by no means complete:

2 Requirements

In the following the requirements for the project are summarized. These requirements concern both the usability from the viewpoint of the user as well as the maintenance of `BreakOut`.

2.1 Real world interaction

As stated in 1.1, one of the main aims of this project was to be able to demonstrate the possibilities of augmented reality. For this reason the user should

AR Quake	http://wearables.unisa.edu.au/projects/ARQuake/www/
AR2 Hockey	http://www.mr-system.com/project/main2e.html
Matchtwo	http://www.cybernarium.de/content_en/welten/spiele/matchtwo
OXO	http://www.cybernarium.de/content_en/welten/spiele/oxo
MR Pong	http://www.mlab.uiah.fi/ kkallio/mr-pong/

Figure 2: Related projects

be able to manipulate the game by interacting with the environment instead of using conventional input methods. Since the user controls the racket in `BreakOut`, the interaction with the racket can be implemented by letting the user manipulate it by moving a tracked object in the field of play.

2.2 Visualization

Unlike most AR projects, `BreakOut` is not intended for use with a head mounted display (HMD). Instead the field of play is to be projected on a flat surface (i.e. a table) by a beamer. The advantage is that although `BreakOut` can only be played by one user it's visibility is not restricted to the user. This would have been the case using a HMD. Additionally by restricting `BreakOut` to a 2D environment the development effort is reduced. This restriction is not detrimental to the game play.

2.3 Configurability

For easier maintenance in the case of future use, `BreakOut` is supposed to be configurable to a large extent without the need of changes to the source code of the project. To realize this configurability appearance and content of `BreakOut` need to be interchangeable solely by modifying or replacing text based configuration files.

3 System design

This section describes the interaction between `BreakOut` and other `DWARF` services regarding the needs and abilities.

The `BreakOut` service only has one need of the type `PoseData`. This need is intended to receive positional data from the racket, supplemented by a tracking device (e.g. the `ARTTracker`). The `ObjectCalibration` service is interposed between the `BreakOut` and the tracking services for calibration purposes. After processing the data of the racket `BreakOut` transmits the processed racket position as well as the ball position over separate abilities of type `PoseData`. During initialization `BreakOut` also transmits data over three additional abilities:

`SceneData`	The actual visualization information
`UserAction`	Necessary for selection of recipients for `PoseData` events
`PoseData`	Used for transmitting the position of the blocks and the viewpoint

All these abilities are sending the necessary information for the visualization. The `Viewer` service can be used for the visualization purpose.

4 Object design

For sake of clarity the name of the relevant functions in the objects are omitted in the description. The focus of the description is the functionality of the objects and not the concrete structure.

4.1 Game logic

Most of the programming went into the development of the classes of this category. All game related calculations are performed here.

BreakOut This is the core component of the service. It is responsible for the initialization and starting of the game as well as keeping track of the progress of the game, i.e. switching to the next level if all blocks are destroyed. For this purpose it stores game related data, consisting of the position and properties of the blocks and general level information. The initial values for the data are parsed using an instance of `LevelXMLHandler` from files on the hard disk.

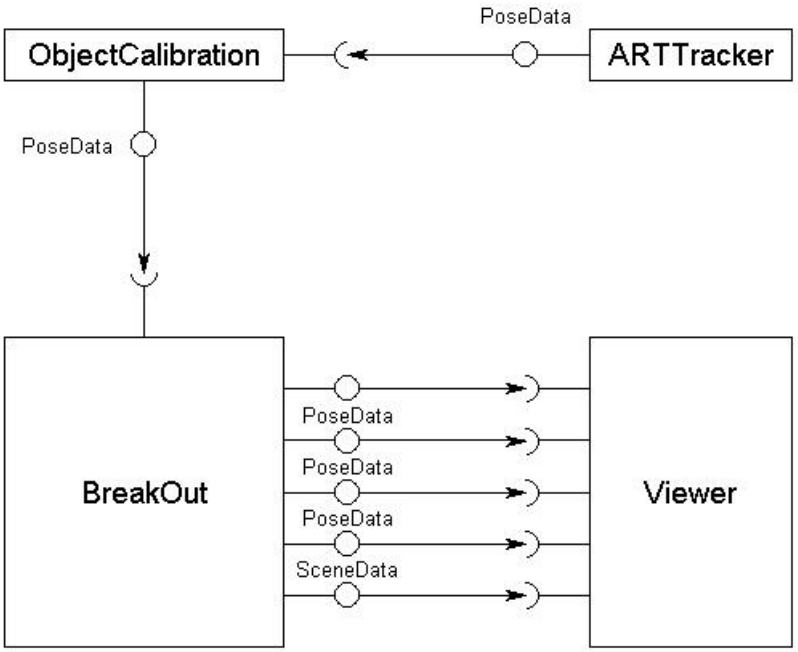

Figure 3: System Design

The instances of the other game logic related objects, the `Racket` and the `Ball` class, are also created in this object. The `Racket` class is used to process the positional data of the real world object representing the racket. This positional data is supplied by the `RealObjectPose` class, which handles incoming `PoseData` events from the tracking component. The `Ball` class calculates the movement of the ball.

Lastly this class communicates with the other `DWARF` services. for this purpose it creates several instances of objects derived from `BasicSender` for transmission of `PoseData` events (for the position of the racket, the ball,the blocks and the viewpoint), `SceneData` events and `UserAction` events and one instance of `RealObjectPose`, which was mentioned above. The `RealObjectPose` and the sender instances correspond to the need and the abilities of this service respectively.

Ball This object is created by `BreakOut`. It's purpose is to calculate the movement of the ball. Before the `Ball` class transmits the current position of the ball it retrieves the position of blocks in close proximity from `BreakOut` and the current position of the racket from the `Racket` class to check wether the calculated movement of the ball results in a collision with a block or the racket. If this is the case the collision has to be resolved before the position of the ball can be transmitted. After finishing the calculations it uses the instance of `PoseDataSender` created by `BreakOut` to send the updated position of the block to the Visualization component (e.g. `Viewer`). If the ball collided with a block in its last movement the `Ball` class notifies `BreakOut` that the concerned block has been hit. In figure 5 you can see a sequence diagram of the flow of action in the ball class.

Racket This object is created by `BreakOut`. It retrieves the positional data of the real world object representing the racket from the `RealObjectPose` class. This positional data is then processed and sent to the Visualization component via the `BreakOut` class.

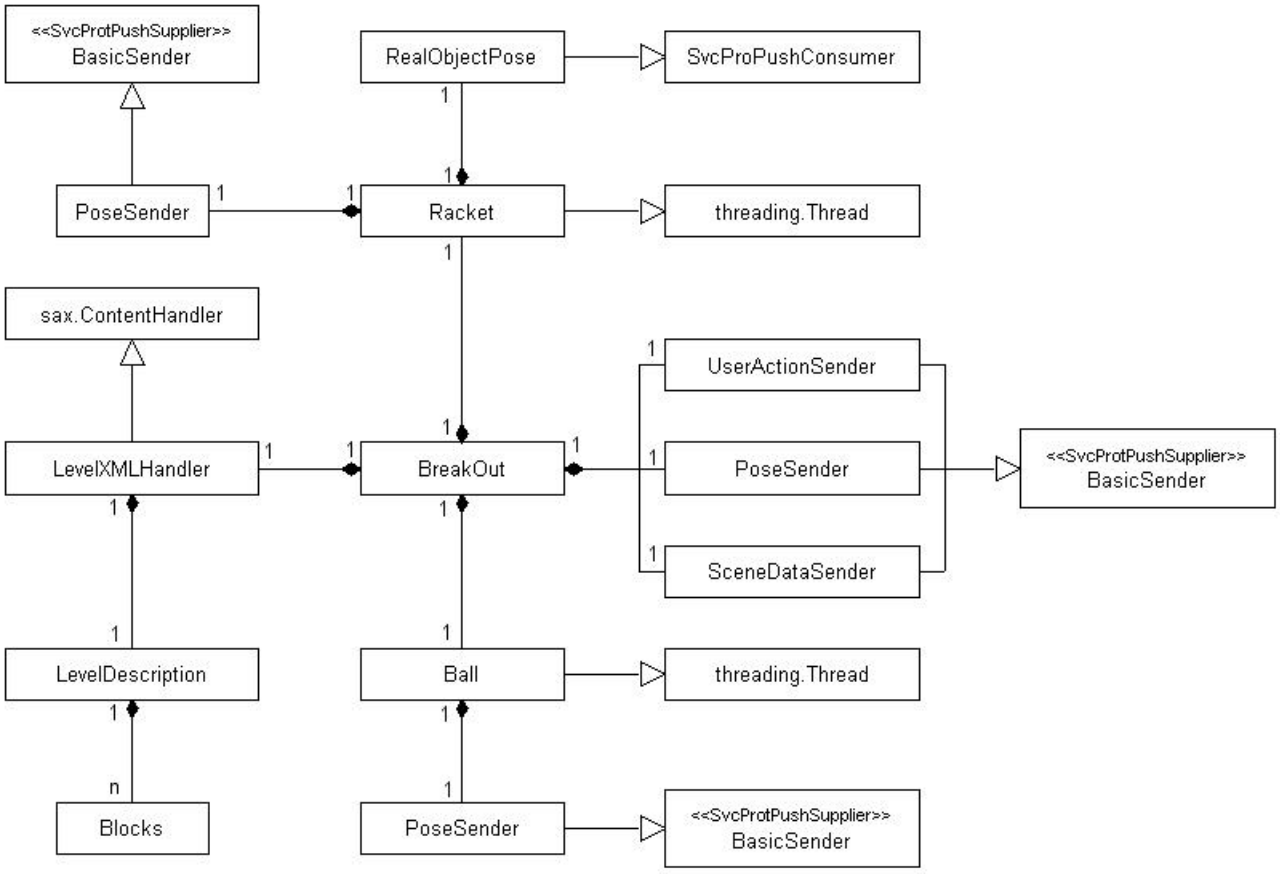

Figure 4: Object Design

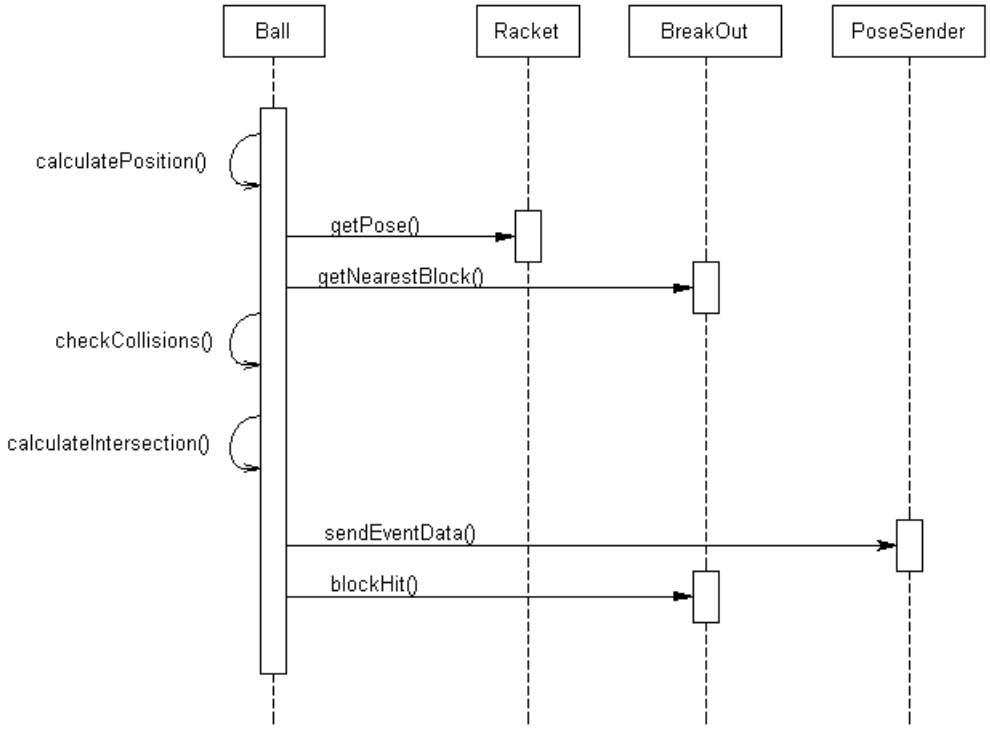

Figure 5: Sequence diagram of the ball class

4.2 Event handling classes

As `BreakOut` is designed as a `DWARF` service communication with other services is based on `CORBA` events. The classes of this category are designed to either handle incoming `CORBA` events or to send these events to other services.

BasicSender This is the base class for all event sending classes. It implements the functions required to maintain the connection to the consumer, i.e. the visualization component, and for sending the event data. Classes derived from this base class only need to implement functions concerning the creation of the event data.

PoseSender This subclass of `BasicSender` is used for sending `PoseData` events. `BreakOut` uses three instances of this class: One for the ball position, one for the racket position and the last one during initialization for the position of the blocks and the viewpoint.

SceneSender This subclass of `BasicSender` is used for sending `SceneData` events. During initialization `BreakOut` sends events of this type to the visualization component for the ball,racket and blocks so that these objects are displayed. During the game it is used to send events removing blocks from the game.

UserActionSender This subclass of `BasicSender` is used for sending `UserAction` events. These events are needed to select objects so that `PoseData` events can be sent to change the position of these objects.

RealObjectPose This class receives the positional data for the real world racket from the tracking component. This data is stored in the class and is later retrieved by the Racket class.

4.3 Data import

To fulfill the requirement of configurability `BreakOut` was designed to parse the relevant information for the game levels from an arbitrary file(see 7.2 and 7.3 for details). For this purpose `BreakOut` makes use of the `xml.sax` package (Simple API for Xml).

LevelXMLHandler This class is a subclass of `xml.sax.handler.ContentHandler`. During one parsing run it stores all information for the parsed level in an instance of `LevelDescription`. This information can then be accessed by `BreakOut`. During the first parsing run it creates a list of all available levels in the parsed file.

LevelDescription An instance of `LevelDescription` stores all relevant information for one level. Amongst other things it contains an array of Block instances containing information about all blocks of the level.

Block This class stores the information for one block in the game.

5 Game physics

Most important aspect is the analysis of collisions. `BreakOut` only checks if the ball collided with the racket or the blocks. This is sufficient since the only other moving object, the racket, is restricted to the bottom of the playing area, which prevents it from colliding with the blocks.

5.1 Collision detection

The first step is to detect wether there actually occurred any collisions. To accomplish this `BreakOut` compares the position of the ball to the position of all other objects, i.e. to the position of the racket and the blocks.

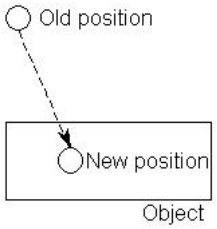

Figure 6: Collision on the top side of object

If the distance between two objects is too small `BreakOut` assumes there was a collision. The check is performed by using the following equation, where

d stands for the distance between the ball and the object. Additionally, r stands for the radius of the ball while s stands for the size of the object.

$d \leq r + s$

This check is executed twice, once for the horizontal and once for the vertical distance/size. If both return true as result `BreakOut` assumes that the ball and the object collide. If `BreakOut` detected a collision it tries to determine on which side of the object the ball collided with the object. To do this it takes into account the position of the ball before the last move. To determine the side `BreakOut` performs four tests, two for the horizontal and two for the vertical positions/size. In those equations p_{old} stands for the position of the ball before the last move, p_obj for the position of the object, r for the radius of the ball and s for the size of the object.

If this test returns true, `BreakOut` assumes a collision on the right or respectively top side of the object:

$p_{old} - r \geq p_{obj} + s$

If this test returns true, `BreakOut` assumes a collision on the left or respectively bottom side of the object:

$p_{old} + r \geq p_{obj} - s$

If the old position for example was above the position of the racket `BreakOut` would assume a collision on the top side of the racket (see figure 6).

6 Collision resolution

If there is a collision it has to be resolved, so that the ball and the relevant object don't occupy the same space any longer. This resolution is carried out in two steps. First step is to move the ball to the position where it touches the object for the first time. The new position for the ball is determined by calculating the intersection of the balls path and the corresponding side of the object. To take into account the fact that the stored position of the ball is actually its center the radius of the ball is added to the position of the considered side (see figure 7). To calculate the intersection `BreakOut` uses the position of four points, while every two points are used to calculate one vector. The first vector, representing the movement of the ball, is calculated from the current position of the ball and the position of the ball before the last move. The second vector is calculated from the edge points of the side of the object, which is considered for the collision.

First of all `BreakOut` calculates the slope of the two vectors:

$s1 = \frac{y2-y1}{x2-x1}$

$s2 = \frac{y4-y3}{x4-x3}$

Then it calculates the zero-crossing of the y axis:

$z1 = y1 - x1 \cdot s1$

$z2 = y3 - x3 \cdot s2$

After these steps it calculates the actual position of the intersection:

$xi = \frac{z1-z2}{s2-s1}$

$yi = s1 \cdot xi + z1$

Second step is to update the flying direction of the ball, to prevent it from colliding with the same object again and to give a realistic impression of a bounce. To determine the new direction `BreakOut` uses one of two different formulas, depending on the orientation of the side from which the ball bounces. For horizontal surfaces `BreakOut` uses the following formula, where d stands for the direction of the ball:

$d = (180 - d)\%360$

For vertical surfaces it uses a different one:

$d = -d\%360$

`BreakOut` uses these formulas for all calculations with only one exception: If the relevant side is the top side of the object and the object is the racket, the calculation is treated differently.

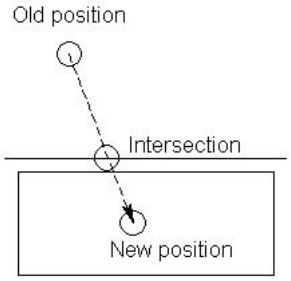

Figure 7: Calculated intersection

6.1 Racket

As the racket has a square shape for the sake of a less complex collision detection, the problem arose that the player could not influence the direction of the ball. Since the ball would only ever collide with

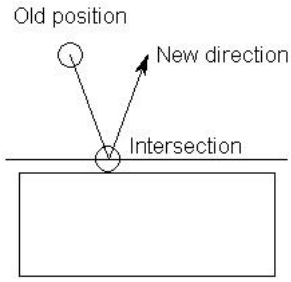

Figure 8: Updated direction

straight surfaces there would only be a few possible directions for it. To counter this problem the top side of the racket is treated as curved. Instead of a constant 90 degrees `BreakOut` calculates an angle ranging from 60 degrees at the left edge to 120 degrees at the right edge. To calculate the actual angle `BreakOut` uses the following formula:

$(\frac{p_b - p_r}{s_r} \cdot 60) + 90$

In this formula p_b stands for the position of the ball, p_r for the position of the racket and s_r for the size of the racket. If the result is greater than 70 and lower than 290 degrees it is changed to the nearer one of those two values. This is so to prevent illegal values as well as values which would result in boring gameplay. Illegal values are values where the updated direction of the ball would cause it to immediately collide another time with the racket. These values can occur due to the curved nature of the racket.

7 Installation & configuration

This chapter deals with everything that has to be done to get `BreakOut` up and running. It also discusses the possibilities of configuration of the game.

7.1 Installation

To install `BreakOut` you first have to install `DWARF`. You can either start the "configure" script during installation of `DWARF` with the parameter "–enableBreakOut" or you can manually install the required files. To do this you have to start "make" in the directories "src/services/BreakOut" and "applications/BreakOut" after installation of `DWARF` is finished. The first call of make installs the required files for the service `BreakOut`, while the second call installs the following files:

Level description A file containing the description of a default set of game levels(see 7.2 for details)

Graphics Several files in the Vrml format containing the graphic elements of the game(see 7.2 for details)

Service descriptions The xml descriptor files for other required services (see 7.3 for details)

These files are copied to the directory "share/BreakOut/".

7.2 Game configuration

`BreakOut` was designed for maximum configurability without the need of changes in the source files. Therefore `BreakOut` parses a number of easily modifiable or replaceable configuration files on startup.

Appearance `BreakOut` uses files in the `Open Inventor` file format for visualization purposes. During initialization of the game several files in this format are being loaded. These files can contain wildcards instead of float integers for the values of certain `vrml` nodes. These wildcards will be replaced either with values from the level description (see 7.2 for details) or with default values.

In the following example the strings "green", "red" and "blue" would be replaced with the respective values from the level description. These wildcards for the colour of an object can be used in all `vrml` files.

```
Material {
    diffuseColor green red blue
}
```

`BreakOut` loads `vrml` files for the following objects. The paths of these files can be configured in the level description file (see 7.2).

Ball Besides the wildcards for the colour the `vrml` file for the ball can contain the wildcard "radius" for the radius of the ball. The default path for this file is "../share/BreakOut/ball.wrl".

Racket This file can contain the wildcards "xSize", "ySize" and "zSize" for the corresponding size values in the geometry node, as well as the colour wildcards. Default path is "../share/BreakOut/racket.wrl".

Block This vrml file is used for all blocks in one level. It can contain the same wildcards as the racket. Default path is "../share/BreakOut/block.wrl".

Playfield This file is used as a background for the playing area. Like in the other files it supports wildcards for the colour. But it does not support wildcards for the position because `BreakOut` always assumes that the bottom left corner of the playing area is at the point of origin. It replaces the wildcard "boundary" with a list of coordinates of the four edges of the playing field. This wildcard is intended to be placed in the "Coordinate" node of an `IndexedFaceSet`. After the "boundary" wildcard there can be an additional wildcard "brim", which is replaced with the coordinates of the points for a simple brim of the playing area. These coordinates are generated from the values of the element "playfield" in the level description file (see 7.2 for details)

Layout During initialization `BreakOut` parses the `levelFile` (see 7.3 for details), which contains the description of all levels in the game. If this file is missing (or a wrong path was passed) then `BreakOut` is unable to start. The description of a level contains amongst other things information about the colour and size of objects. These information can only be applied in the visualization if the corresponding `vrml` files contain wildcards for the respective values (see 7.2 for details). The description is implemented in a `xml` notation. On page 10 you can see a list of all supported elements in this notation.
With the exception of the block elements any element can be omitted. If an element is missing from the description of a level `BreakOut` uses default values for all of the elements attributes. The same applies to missing attributes. If one or more attributes of an element are missing `BreakOut` uses default values. Only in the block element all attributes other than zPose have to be specified. For zPose `BreakOut` assumes the value "0" since it is intended for use in a 2d environment.
In figure 11 you can see a simple example of a `levelFile`: It contains only one level called "Level 1". This level has a ball which is slightly larger and faster than the default ball and also has a different colour. There are three blocks in this level, which are arranged in a line in the middle of the playfield. The middle one of these three blocks needs to be hit two times to be destroyed, while the other are removed immediately.

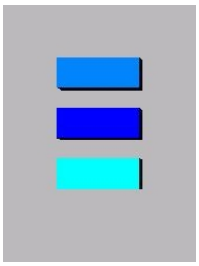

Figure 10: Differently coloured blocks depending on `hits` attribute

7.3 Starting BreakOut

To start `BreakOut` you just have to execute the file "BreakOut.py" in the "bin" directory.

Command line parameters At the moment there are two command line parameters which can be passed when executing "BreakOut.py".

levelFile This denotes the path of the file from which `BreakOut` reads the descriptions of all the levels in the game (see 7.2 for details). If this parameter is not specified `BreakOut` will try to read the file from the path "../share/BreakOut/BreakOut_Levels.xml"

startLevel This parameter can be passed to specify a level, which will be loaded at the beginning of the game. If this parameter is not specified, `BreakOut` will determine a random level which will be used as the `startLevel`.

`breakout`	The `SAX xml` parser, which is used for parsing the `levelFile` requires an enclosing group element in the `xml` document. This element has no further meaning, it just has to be present.
`level`	This element encloses all of the elements of one level. A level is identified by the "name" attribute of this element.
`ball`	All the relevant information about the ball in the level are stored in this element. Possible attributes: `id` id of the ball (string) `speed` speed of the ball (float) `radius` radius of the ball (float)
`ballColour`	The colour of the ball. Possible attributes: `green` green colour value (float) `red` red colour value(float) `blue` blue colour value (float)
`ballFile`	The only attribute of this element is "path", which specifies the `vrml` file which shall be used for representation of the ball.
`racket`	The attributes of this element define the size of the racket in this level. Possible attributes: `length` length of the object (float) `width` width of the object(float) `height` height of the object (float)
`racketColour`	analog to `ballColour`
`racketFile`	analog to `ballFile`
`playfield`	Specifies the size of the playing area. These values are used for the game logic as well as for the customization of the visualization file of the playfield. Possible attributes: `length` dimension on the x axis (float) `width` dimension on the y axis (float) `height` dimension on the z axis (float)
`playfieldColour`	analog to `ballColour`
`playfieldFile`	analog to `ballFile`
`blockSize`	analog to `playfield`
`blockColour`	The colour values are configured analog to `ballColour`. Additionally there is an "offset" value for each colour values. This offset is added to the colour value of a block if its `hits` attribute is set to two, and added twice if its `hits` is set to three or more. Possible attributes: `offsetGreen` added to the green colour value depending on hits (float) `offsetRed` added to the red colour value depending on hits (float) `offsetBlue` added to the blue colour value depending on hits (float)
`blockFile`	analog to `ballFile`
`viewpoint`	The position of the viewpoint for the scene. Possible attributes: `xPose` position on the x axis (float) `yPose` position on the y axis(float) `zPose` position on the z axis (float)
`block`	The description can contain any number of elements of this type. Each block, which should appear in the level needs to be listed in the description. Required attributes: `id` id of the block (string) `xPose` position on the x axis (float) `yPose` position on the y axis(float) `zPose` for future use, can be omitted `hits` amount of hits needed to destroy block(integer)

Figure 9: Supported elements in the `xml` notation for `levelFile`

```
<breakout>
  <level name="Level 1">
    <ball speed="0.03" radius="0.02" />
    <ballColour green="0.2" red="0.8" />
    <block id="Block1" xPose="0.4" yPose="0.6" hits="1"/>
    <block id="Block2" xPose="0.5" yPose="0.6" hits="2"/>
    <block id="Block3" xPose="0.6" yPose="0.6" hits="1"/>
  </level>
</breakout>
```

Figure 11: Example for `levelFile`

So if you want `BreakOut` to start with the level "Test", the command would look like this:
`./BreakOut.py "startLevel=Test"`

Other required services `BreakOut` is dependent on a few other services. In theory `BreakOut` works with any services which have the required `Needs` and `Abilities`, but it is intended to be used in conjunction with the following:

- `Viewer`: A visualization service based on the `Open Inventor` file format.
- `ARTTracker`: A tracking service processing the data sent by the `ART` tracking device.
- `ObjectCalibration`: A calibration service.

These services have to be manually started to get `BreakOut` into a working state.

8 Results

This chapter provides a conclusion about this project and the suitability of the deployed tools as well as a summary of possible future work on the project.

8.1 Conclusion

During the course of this project a working implementation of the `BreakOut` game concept has been finished. This implementation has been designed as a `DWARF` service. It relies on other services for parts of its functionality, namely a tracking service and a visualization service. According to the requirements appearance and content of the game are customizable without changes to the source code.
Overall the applied tools varied in their suitability for this project.

DWARF This project has been developed as a component within the `DWARF` framework. Using this framework has led to some performance problems due to the distributed layout of the components, mostly in the form of time delays occurring between the `BreakOut` and the `Viewer` service . At the same time `BreakOut` does not make use of the distributed properties offered by the `CORBA` architecture. Still, using `DWARF` offered an easy access to existing services, particularly the `Viewer` and the `ARTTracker` services. These two services were used for all visualization and tracking purposes respectively.

Python `Python` was used for the implementation of all classes in this project. `Python` is an easy to learn, flexible language with a clean syntax. It's interactive mode can be very helpful during development and testing of software, as it reduces the need for compiler runs speeding up the development process. Only the lack of type checking needs getting used to.

ARTTracker The transmission speed of positional data by the `ART` tracking device and service was usually within an acceptable range. A problem is that players can obstruct the field of view of the cameras of the tracking device resulting in a disruption in the stream of positional data. If this happens the movements of

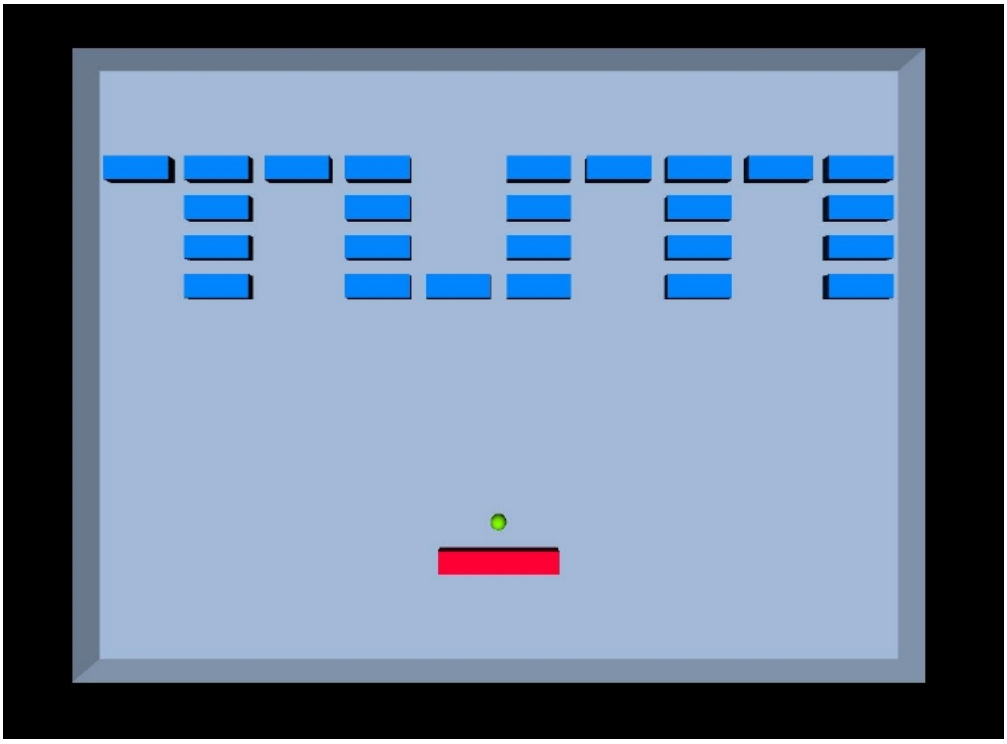

Figure 12: Implementation of BreakOut

the racket by the player can't be processed anymore. Still it was overall a reliable choice.

8.2 Future work

After completion of this project there are still aspects of the game which could be improved. Additionally some work could not be finished due to the status of other projects.

8.3 New features

Gameplay of `BreakOut` could be improved by the implementation of additional features. Possible would be for example the addition of a two player mode. Another thinkable feature would be blocks which drop "power-ups" upon destruction, which could then be collected by the player by moving the racket on these objects. These features were omitted to further reduce the development effort.

Viewer adaptations At the time the development was closed the current version of the `Viewer` component, which was used for visualization purposes, had some limitations. Work on a new version was in progress but not finished yet, forcing `BreakOut` to use some work arounds for existing problems. At the moment `BreakOut` sleeps for 0.5 seconds after every event sent to the `Viewer`, as problems can occur if the `Viewer` receives too many events within a short time span. Additionally, it was impossible to quickly change some properties of objects in the `Viewer`. Since the colour of a block in `BreakOut` depends on the amount of hits it can withstand before destruction, it would be desirable to be able to change the colour of a block after it has been hit. Due to the limitations of the `Viewer` it was however impossible to quickly do so and therefore the colour is left the same. These work arounds can be removed once the new version of the `Viewer` is completed.
Due to the excessive changes in the `Viewer` component it is possible that the new version is not downward compatible.If this is the case additional modifications to `BreakOut` would be necessary to render it compatible with the `Viewer` component.