da614a - Connected Systems and Devices

Project Report

Asymmetric Augmented Reality Game

2017-01-10

Alexander Baldwin
Simon Gullstrand
Björn Hansson
Johan Holmberg
Jonas Wahlfrid

Introduction

With the advent of a new generation of hardware such as HTC's Vive[1], Microsoft's Hololens[2] and the Oculus Rift[3], virtual, augmented and mixed-reality applications are experiencing a dramatic increase in popularity[4]. Augmented Reality (AR) provides a view of the physical world that is augmented or enhanced by the use of additional audio or visual information. Due to their tight connection to the physical world ("reality"), AR applications can benefit greatly from the use of context awareness. A context aware system is able to modify its behavior to physical or virtual changes that may occur in the environment and operational context where it is installed as well as changes to the system itself or its users. Contextual cues could make an AR application feel more relevant or realistic by using, for example, elements detected in a video image as part of the application's functionality.

Both AR and context awareness[5] have been important topics of research related to the Internet of Things (IoT)[6]. IoT encapsulates the idea that almost any device that can or could benefit from an internet connection will be or already is connected. The trend can be explained by the vision "Anytime, anywhere, anymedia, anything"[7] which has for a long time been the driving force for the advances in communication technologies.

As an investigation of these topics, we developed a novel application consisting of a number of connected software and hardware components. Asymmetric Augmented Reality Game (AARG) is a hectic, high-pressure AR game where two players with asymmetric roles must communicate to defuse bombs before the time runs out! The game uses context-aware AR elements as the core of its functionality, in combination with a web service-based architecture for facilitating communication between devices. Choosing to use these methods as part of a game gave us an opportunity to see techniques commonly used in IoT systems within an alternate setting.

In this report, we describe the process of designing and implementing AARG. This is followed by a discussion of the main challenges we faced when working on the project and a reflection on alternative approaches we could have taken and how the system could be extended in the future.

[1] https://www.vive.com/eu
[2] https://www.microsoft.com/microsoft-hololens/en-us
[3] https://www3.oculus.com/en-us/rift
[4] Anders Henrysson and Mark Ollila. 2004. UMAR: Ubiquitous Mobile Augmented Reality. In Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia (MUM '04). ACM, New York, NY, USA, 41-45.
[5] Maria Danninger and Rainer Stiefelhagen. 2008. A context-aware virtual secretary in a smart office environment. In Proceedings of the 16th ACM international conference on Multimedia (MM '08).
[6] L. Atzori and A. Iera and G. Morabito. The Internet of Things: A survey. Computer networks, 54(15), 2010.
[7] JUN, Zhang, et al. The internet of things. IEEE Commun. Mag, 2011, 49.11: 30-31.

Design and Implementation

Here, we describe how the system was designed and implemented - starting with the original concept and its evolution, before discussing each individual component in detail. For a better understanding of how the final prototype functions, please refer to our demo video[8].

Conceptualisation

The project was based on a concept proposal submitted before implementation began. The concept evolved through group discussions, taking inspiration from several separate ideas including an augmented reality puzzle game loosely based on Lemmings[9] and an application for keeping track of stock levels in supermarkets using image processing techniques. Our original concept is described below (adapted from our original proposal), followed by the changes we made to the concept early in the implementation phase.

Original Concept

Our original proposal describes a two player AR game. Player one controls an Axis Q6045-E camera with Pan Tilt Zoom (PTZ) functionality that is positioned in the middle of a room (see figure 1) and used to scan the room for bombs, which are randomly positioned in a ring around camera and overlaid on the camera image (see figure 2). Each bomb has a timer and must be defused before it explodes. Player two receives instructions from player one about where the bombs are located and must walk around the room to these locations and perform an as-yet undefined interaction in a smartphone app to defuse each bomb before the time runs out.

Communication between the players can take one of two forms:

1. Player one and player two are in the same room and communicate verbally - the approach intends to create frantic and fun interactions between the players as the time pressure builds.
2. Player one and player two are not in the same room. Player one communicates with player two using a limited set of commands sent through the the camera control UI to player two's aforementioned smartphone app (e.g. go left, go right).

Puzzle elements can also be included, such as defusing the bombs in a particular order etc.

[8] https://www.youtube.com/watch?v=H2Q0k6yuxPE
[9] https://en.wikipedia.org/wiki/Lemmings_(video_game)

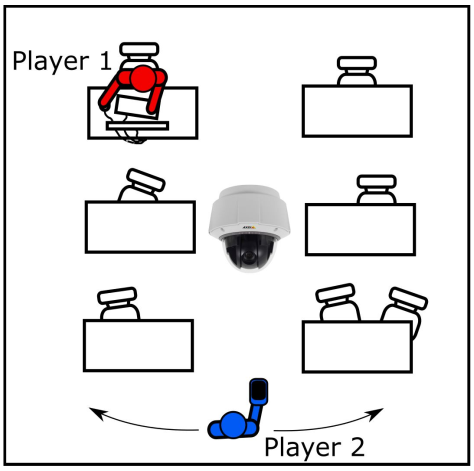


Figure 1. Player one controls the camera from a computer (or possibly a smartphone), scanning for bombs. Player two must move around the room at player one's behest, to defuse bombs.

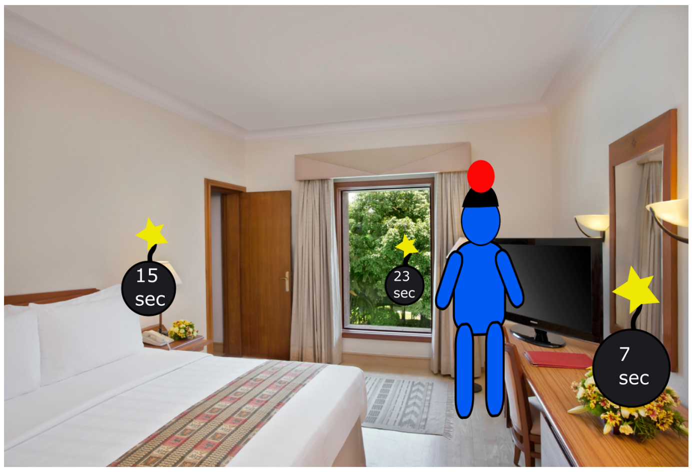


Figure 2. Player one's view. Bombs are overlaid on the camera's image. Player two is tracked either using the existing Axis Digital Autotracking API or through other image processing techniques (such as tracking the colour of an item of clothing the player is wearing).

The concept describes two separate applications (in an architecture shown in figure 3):

1. A Java application for controlling game logic, video streaming, camera controls and communication between players.
2. An Android application for defusing bombs and displaying the game's status.

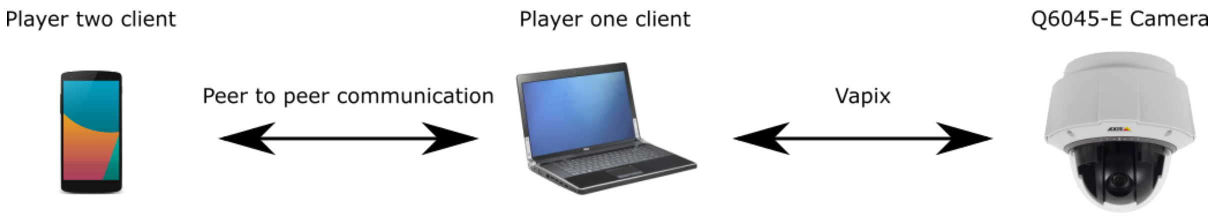


Figure 3. The system architecture proposed in our original concept description

Changes to the Concept

Moving from the proposal phase to the start of implementation, we made some changes to the original concept based on an initial investigation of the scope and implementation challenges associated with the project.

In order to limit the project scope to allow us to implement it in the available time, we decided to restrict development to core functionality and leave additional functions to be implemented if time allowed. As such, we decided not to include puzzle elements in the bomb defusal process and restrict player to player communication to verbal, rather than integrating it into the client and server applications. We also felt that these decisions were appropriate from a gameplay perspective, since it would allow us to create a more streamlined playing experience.

A number of technical decisions were also made before implementation started. We decided to use an HTML and JavaScript-based web application instead of an Android application for the client (the application running on player two's smartphone). Our main reason for this decision was the experience level of the group members and the platform independence of web applications, making for easier testing and a wider potential user base.

A quick investigation of the AXIS Digital Autotracking API showed that it would require serious changes in the gameplay design to use, so we decided early that we would prioritise an image-processing (colour-tracking) based approach to tracking player positions. We identified OpenCV[10] as a potentially suitable image processing library.

Limitations

We have developed a prototype, meaning that the system lacks, by design, many of the features that would be expected in a polished computer game. In particular, there are no real menus and minimal audio and visual feedback. The system is also strongly coupled to a very specific software and hardware configuration: it is not, for example, possible for the game to function with a different type of camera (even one with PTZ functionality). In other areas though, the system is flexible, since our use of a web service for communication means that alternative clients, such as an Android application, could be written without altering the server application.

System Components

The system consists of two applications: a Java-based server (intended for use on a PC) and a JavaScript-based client (intended for use on a smartphone). The client application is responsible for displaying the game's state to one of the players, which it keeps updated by repeatedly polling the server for updates. Figure 4 shows an overview of the system architecture.

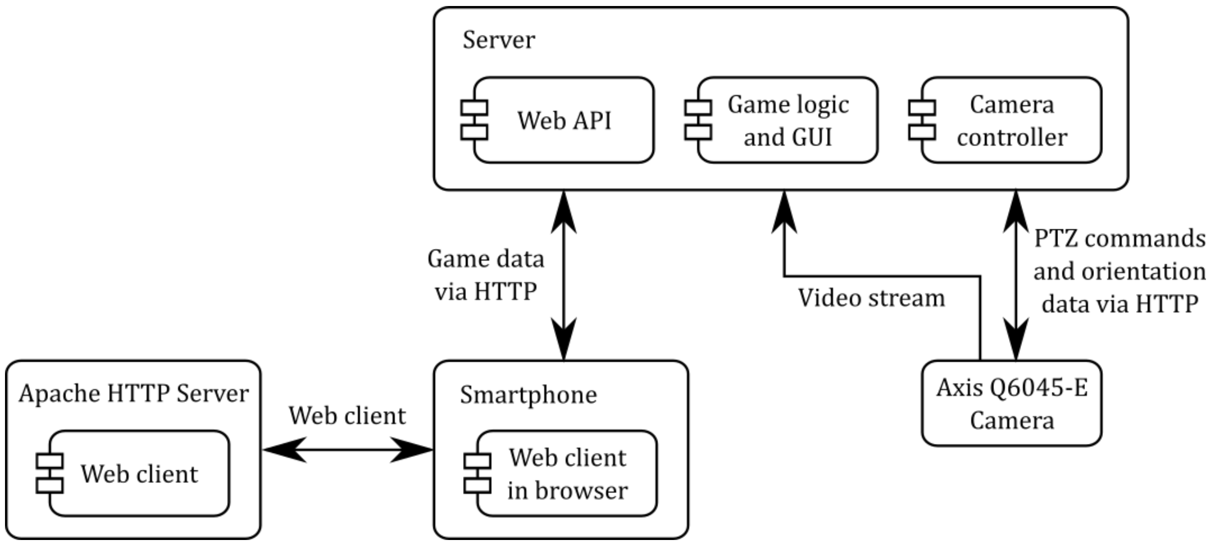


Figure 4. System architecture

[10] http://opencv.org

Server & Core Game

The server application is divided into three components: the first is responsible for the game's logic and GUI, the second is responsible for facilitating communication between server and client and the third is used to control the Axis Q6045-E camera's PTZ functions.

Web API

The web API is developed to enable communication between the involved devices in the game. It provides a REST[11] interface and uses JSON[12] as data format. Exposing functionality through a web API means that an application consuming said functionality can be realised in more than one way. One example of this could be to use a physical defuse button, rather than a smartphone-based game client. A well-defined API will also act as a contract between a server and a client application, potentially simplifying the continued development of the involved software components.

Before writing any code, Apiary[13] was used to document the REST interface. This was then used every step of the way to implement the corresponding API. The Web API is written in Java and implemented with the help of Spark[14], which is a microframework for web applications. To make the code comprehensible we decided to break it down into different models for each separate call type. As seen in figure 4, the game logic and the API are running on the same machine (and as part of the same application). The API is initialised by the game logic class which is the main controller of the game.

As it was created at a very early stage in the development process, the API model ended up exposing functionality that was never implemented in the final product. This unimplemented functionality include e.g. initializing a game from the client, support for more than bomb defuser, and persistent game statistics. These functions enable the game to be extended in the future, but might also be seen as unnecessary in the context of a prototype.

The REST API was designed with Richardson's maturity model[15] in mind. While having an API that fully complies with the Hypermedia As The Engine Of Application State (HATEOAS) would be interesting, it was deemed to be too much of a challenge to actually implement this. Thus, what was designed makes use of the majority of HATEOAS's core concept, such as descriptive links, but leaves out some of the more complex concepts, e.g. non-typed resources. The resulting web API is available at our Apiary page[16].

[11]Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)".*Architectural Styles and the Design of Network-based Software Architectures* (Ph.D.). University of California, Irvine.
[12]http:// json.org
[13] https://apiary.io
[14] http://sparkjava.com
[15] http://martinfowler.com/articles/richardsonMaturityModel.html
[16] http://docs.aarg.apiary.io

Core Game - Gameplay Logic

Game flow is controlled by a simple finite state machine and is divided into 3 core phases illustrated in figure 5.

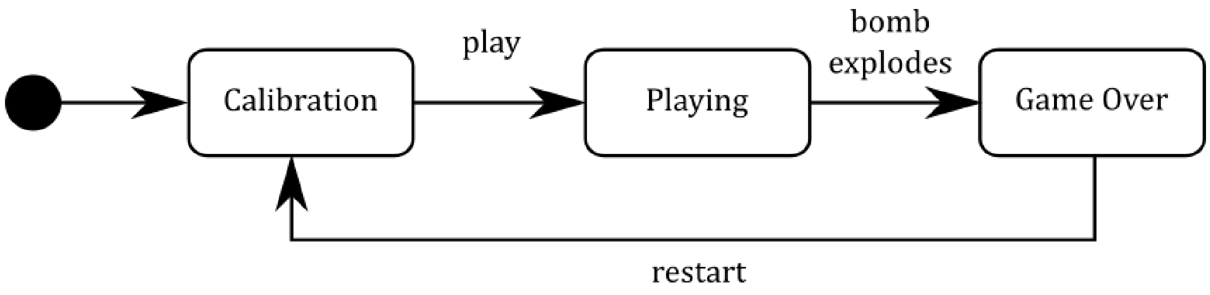
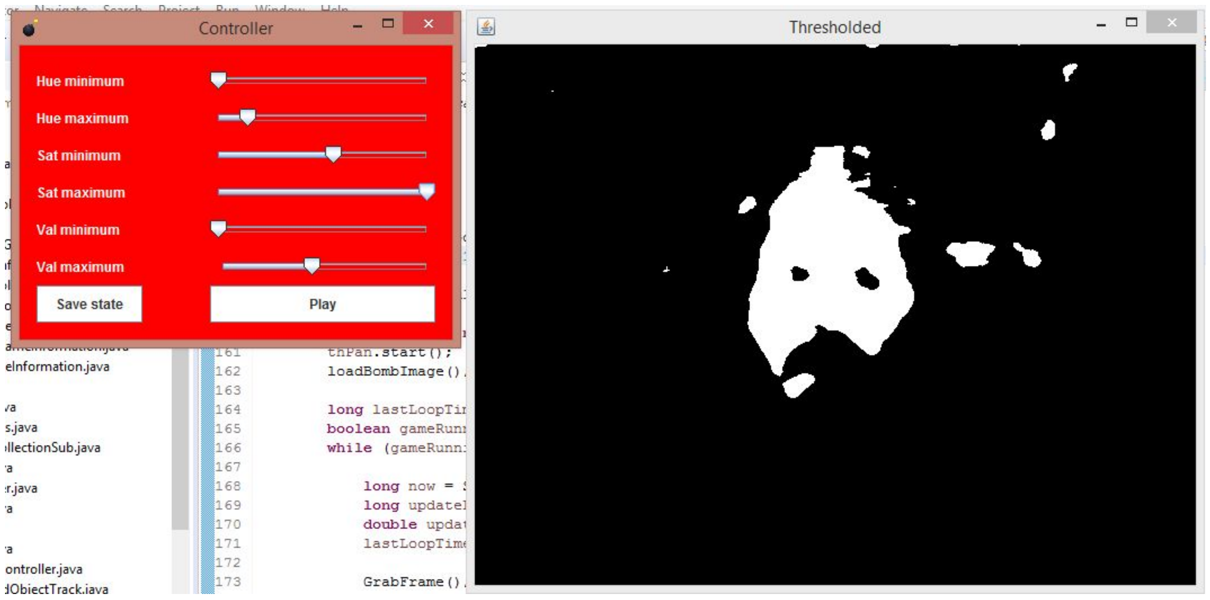


Figure 5. State diagram showing the possible game states.

The *calibration* phase allows the players to define the colour that will be tracked (the process for this is described in the next section) - ideally the colour of a garment worn by player two. This calibration is necessary since changes in lighting conditions can make the same object appear very different between multiple play sessions, precluding the use of hard-coded colour ranges. As well as solving this problem, the calibration also allows for more flexibility: instead of, for example, hard-coding that the game should track a red object, the game can adapt to many different colours (though it should be noted that bold, solid colours that are not normally found in the background of a room will make the tracking more accurate).



```
        thPan.start();
        loadBombImage().

        long lastLoopTi
        boolean gameRunn
        while (gameRunn

            long now = 
            long updateL
            double updat
            lastLoopTime

            GrabFrame().
```

Figure 6. The calibration window showing how the colour values are specified. Objects that appear completely white will be tracked (in this case, Alex's face).

The calibration windows are shown in figure 6. By adjusting minimum and maximum values for hue, saturation and value, the user describes a range of colours that will be identified and tracked. This range can be saved to prevent the need for recalibrating between every play session. Once calibrated to the user's satisfaction, the game can be started by clicking "Play".

During the *playing* phase (as shown in figure 7), bombs appear at regular intervals in a 360 degree area around the camera. Each bomb displays the amount of time remaining before it will explode. Player one must rotate the camera left and right using the left and right arrow keys on the keyboard in order to find bombs that are outside of the camera's viewing angle. Player two's position is tracked while he/she is visible to the camera and bombs can be defused when the player is standing close to a bomb (see Web Client and Web API for more information on defusal).

The *game over* phase (as shown in figure 8) is entered when the players fail to defuse a bomb before its timer runs out. All bombs will explode simultaneously and after 10 seconds the player is automatically returned to the calibration phase.

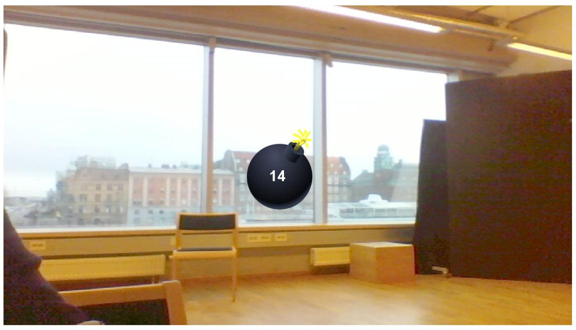


Figure 7. Game screen during the playing phase. One bomb is shown in the center of the screen with 14 seconds remaining before it explodes.

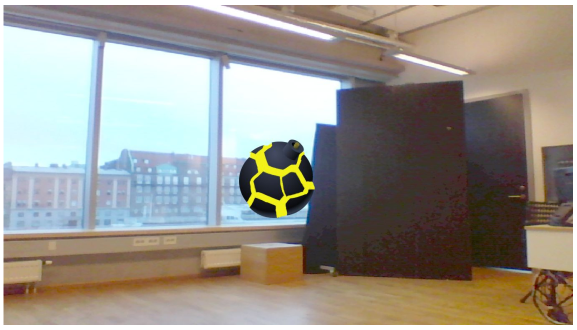


Figure 8. Game screen during the game over phase. A bomb is in the process of exploding.

Core Game - Object Tracking

The function of defusing bombs relies on contextual awareness of the the defusing player's location in the room. We achieve this by analysing the video image to find where the player is located in the camera's field of view and then combine this with the camera's orientation (which is queried from the camera) to calculate an approximate location.

We use JavaCV[17] (a wrapper for the computer vision library OpenCV) to identify the position of a coloured object in the camera image. Using this, we can track the approximate position of a person by calibrating the application to track the colour of an item of clothing the person is wearing. The broad details of the algorithm used (adapted from a JavaCV example project[18]) are as follows :

1. A video frame is captured from the camera
2. Pixels matching the colour range to be tracked are turned white, others are turned black.
3. The image is smoothed to remove noise.
4. The average position of the white pixels is calculated (this represents the approximate position of the tracked object). The result can be seen in figure 9.

[17] https://github.com/bytedeco/javacv
[18] https://github.com/bytedeco/javacv/blob/master/samples/ColoredObjectTrack.java

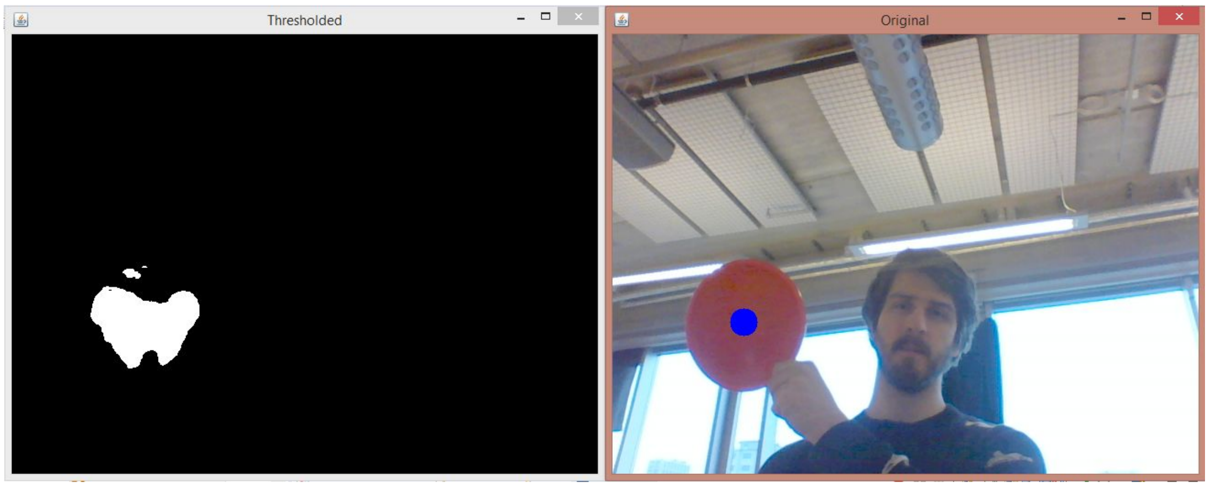


Figure 9. On the left: a video frame after processing. On the right: the identified position of the tracked object (a red plastic lid), shown by the blue circle.

In order to be useful, the screen position must then be put in relation to the location of the bombs and the camera. The "position" of a bomb is described by its angle (the distance from the camera is not considered - only the angle), where 0 is the starting angle for the camera and the bomb positions can be between -180° and 180°. As such, we also translate the tracked screen position to an angle between -180° and 180°. To do this we exploit the fact that the camera we are using has a viewing angle of 62.8°[19] and map the horizontal component of the tracked position to an angle between -31.4° and 31.4° from the camera's angle, as illustrated in figure 10. While this method is not completely accurate (that is, if player two and a bomb are in the same position on screen, they will no longer be in exactly the same position after the camera is rotated), we judge it sufficiently accurate for our purposes since players do not need to be at the exact position of a bomb to diffuse it.

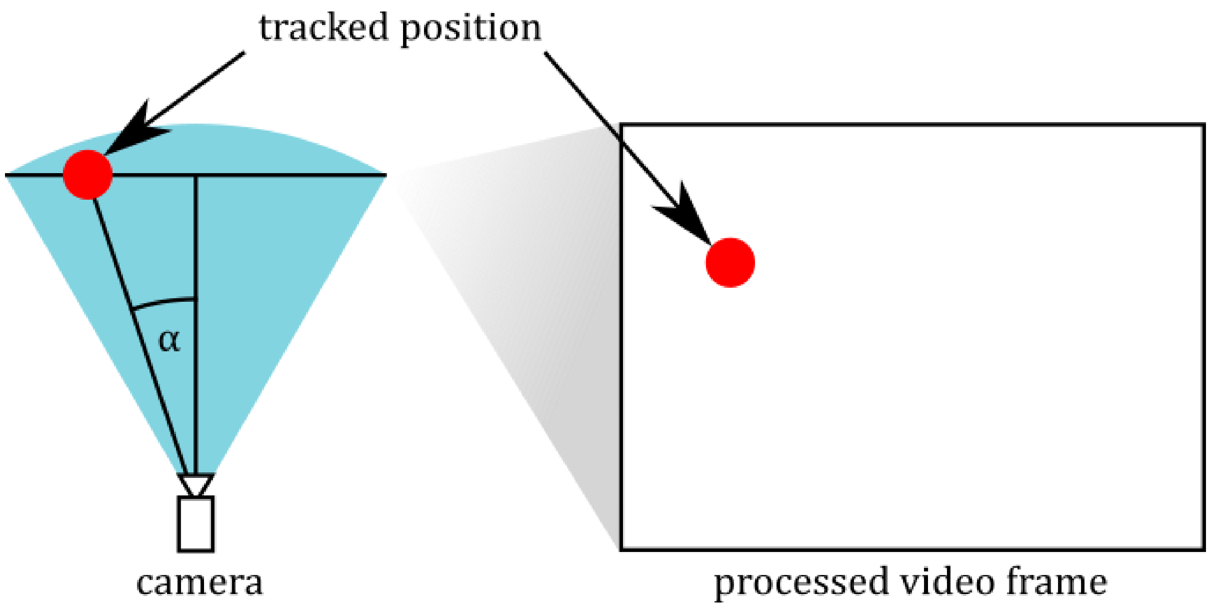


Figure 10. The tracked screen position in relation to the calculated angle of the player. The player is α° from the camera's angle in an anticlockwise direction.

[19] http://www.axis.com/ie/en/products/axis-q6045-e

Core Game - Camera Control

The camera controller was also written in Java to be easily merged with the main logic of the game. Since Axis's own documentation of the VAPIX HTTP API that we use to control the camera's PTZ functions is difficult to use, we mainly relied on an alternative source[20] which was very helpful when implementing the necessary logic to control the camera's movement. The logic works as follows:

- When the class is initialized, it is started on its own thread since it must constantly poll the camera for changes in orientation.
- At startup, three HTTP requests are sent to the camera: enable autofocus, set the tilt angle to 1 degree upwards (which makes the camera approximately horizontal), and set zoom to lowest value (1x). In essence, we set the camera back to a default state to ensure consistency between game sessions.
- The camera starts to pan when the left and right arrow keys are pressed, by means of HTTP requests. Key repeat are filtered out. Similar requests stop the camera movement when the arrow keys are released.
- The controller continually requests the camera's orientation in order to keep it updated in the application so we know where the bombs should be positioned in relation to the video image.

Communication with the camera was realised using the Unirest library[21] for sending HTTP requests.

Web Client

The bomb defuser is equipped with a web interface, typically displayed on a smartphone. This component is realised as a single page web application, using JavaScript for the game logic. The graphical aspects are implemented using Bootstrap[22] for the responsive design abilities and pre-existing stylesheets. jQuery[23] is used to be able to save time when writing JavaScript and get cross-browser support. Communication with the core game's web API uses jQuery's AJAX functionality.

We opted to let the web client follow the game model described in the section *Core Game - Gameplay Logic*, thus leaving out the unimplemented functionality exposed by the web API. By doing this, we were able to slim down the complexity of the web client considerably. The possible states of the web client are shown in figure 11.

[20] http://home.arcor.de/basys/web/CAM/AxisCamera_HTTP_API.htm
[21] http://unirest.io
[22] http://getbootstrap.com
[23] https://jquery.com

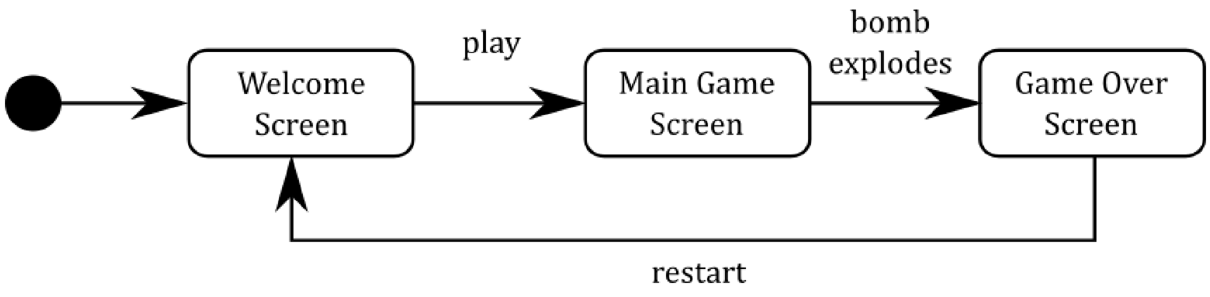


Figure 11. States of the web client

Architecture

The client is roughly realised as an MVC application, the architecture of which is shown in figure 12. The view, consisting of a single HTML document and some CSS is manipulated by a set of JavaScript functions. The games and players are modeled as two JavaScript objects, which are controlled by a game controller object. Finally, all parts are held together by a game flow object. Configuration details are gathered in a common configuration file, which is made available to the rest of the application as a configuration module.

Breaking up the application into discrete modules is made possible by using the RequireJS module loader library[24]. By using this library, we can safely assume that no functionality will break due to issues stemming from asynchronous file loading, which has historically been a common problem in web development. Modularisation of an application provides many benefits, especially in terms of reduced internal complexity.

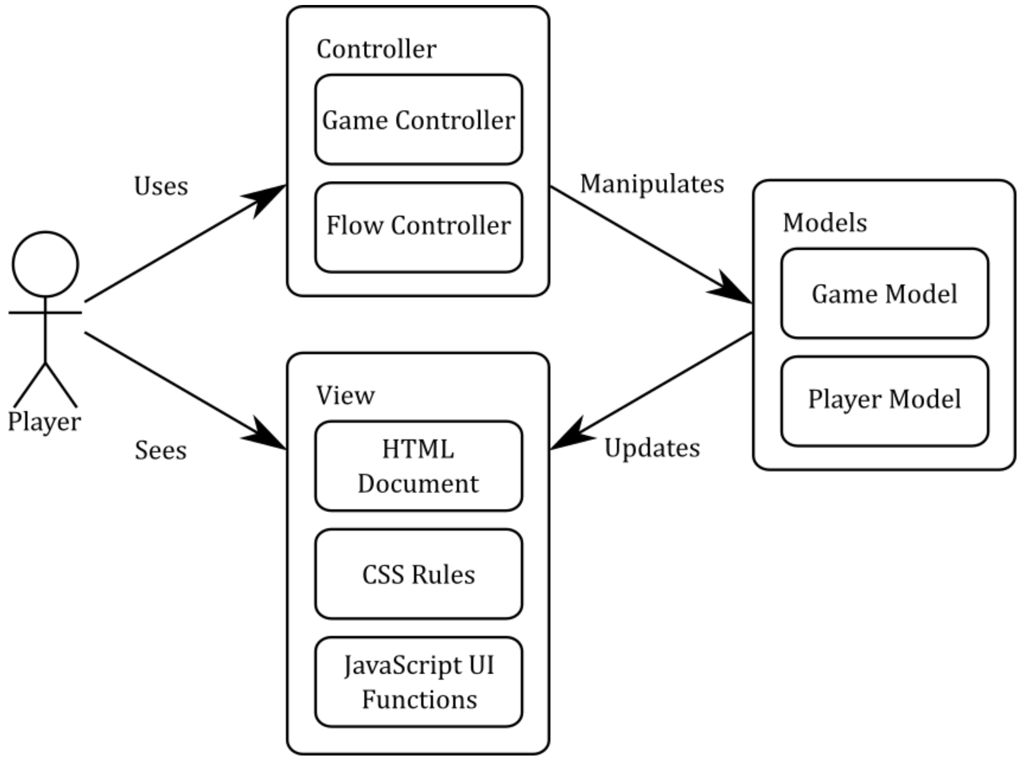


Figure 12. Web client architecture

[24] http://requirejs.org

Communication with the core game

The web client communicates with the core game through the web API described in the *Web API* section. It does so by using the AJAX functionality provided by jQuery. Using jQuery is beneficial for a number of reasons, one of the major points being its ease of use and use of (rather primitive) promises[25]. Promises provide a sane method of handling asynchronous calls, thus making web API usage relatively hassle free.

As previously described, the client doesn't make use of all aspects of the API. This is especially true for the HATEOAS-related aspects of the API. Thus, it needs to have some prior knowledge of what the API looks like prior to the actual communication with the API can commence. This means that the client makes use of out-of-band communication, which ties it to the server in a suboptimal way. One example of this is how the client is tied to a specific game server by defining its address in the configuration file.

Discussion

Naturally, we encountered challenges during the development of AARG and we present some of them in this section, together with a reflection on some other approaches we could have used. Finally, we discuss some of the directions this work could take in the future, if development were to continue.

Challenges

Organisation and Planning

Our communication, planning, requirement-tracking and decision making could all have been improved. Our shortcomings here can mainly be seen in the problems we had when integrating components that had previously been developed separately by one or two people. For example, when integrating the game logic and the web API, we uncovered some fundamental inconsistencies in how we had interpreted the game logic that required a redesign and reimplementation of large parts of the API and, consequently, the web client.

During the project's later stages, when work on integrating the separate components had begun, we started to use Trello[26] to assign and keep track of tasks. Some group members used Trello more than others and there should have been more discussion of the nature of the tasks added to our board to make it easier for anyone in the group to pick up a new task upon finishing another task. Many of our organisational problems could likely have been solved by integrating a project management tool like Trello earlier in the process and by using good practices from a tried-and-tested software engineering method like Scrum. Standup meetings would have kept the group more aware of what everyone was working on and allowed us to identify potential problems earlier and prioritise the most important tasks. Additionally, more

[25] https://www.promisejs.org
[26] https://trello.com

rotation between different tasks could have given everyone a better understanding of the overall status of the project.

For a larger project an agreed way of working with Git, code style, code documentation and test cases would be needed. Now the way of working has been more like “just get it to work”, which can be acceptable for a small-scale project, but as mentioned, was certainly not problem free! We used a Google calendar for planning which days to work on the project. Unfortunately, members did not always follow the agreed time. Furthermore, setting up the camera each day, working from laptops without large screens and not having a dedicated room is not effective.

Web service

Building a RESTful web service in Java is very easy because of all the many high quality external libraries that are available. For example to handle the HTTP communication we used Spark which was simple to use and was well documented. Documenting our web service sounds like a pain but with the tool Apiary we managed with ease. Documenting in advance dramatically simplified the process of implementing the code later. However, as mentioned above, there were problems when integrating the components of the server application, which suggests that there should have been more discussion about how well the documentation conformed to our functional requirements before continuing with implementation. A meeting where the group looked through the documentation together would likely have saved us quite a bit of effort later.

Web client

We decided to to use a web-based JavaScript application instead of e.g. an Android application, because a web client can be used on any type of device and platform, thus not being limited to only the Android platform. Using RequireJS to split up the JavaScript application into discrete modules proved to be a successful approach, but took some time and effort to set up properly.

While most of the development of this component was fairly straightforward, there proved to be some issues in the integration phase. Those issues stemmed mostly from minor bugs in the API and how the API calls were handled by the client. Using the API mock server provided by Apiary helps a lot when developing a client, but not all use cases are supported by the mock server. Apiary also provides support for API verification, but this functionality was not used during the development.

During the final phases of development we experienced some minor hurdles with publishing the web client on the local network that we used for playing the game. While not being strictly related to the development, it should be noted that deployment can sometimes prove to be more complicated than anticipated.

OpenCV and Object Tracking

OpenCV has a steep learning curve, particularly for a developer not familiar with computer vision techniques. This problem only became worse when using a wrapper like JavaCV, since functions don't necessarily have the same signatures and the documentation is less comprehensive. When searching for help online, it was hard to find information that is relevant to specifically JavaCV (and there are at least two totally different wrappers called JavaCV) due to the many wrappers for different languages and the overwhelming focus on the C++ variant of OpenCV.

Our use of JavaCV also resulted in numerous memory leaks: a danger of using wrappers to native libraries. It took a bit of hunting to find the sources of the memory leaks since it's unclear in which cases the wrapper takes care of memory allocation/deallocation for us - we were not able to resolve all memory issues within the frame of this project, but were able to make the application run well enough for testing purposes (but very long play sessions will result in the computer running out of memory). Since JavaCV doesn't have a very big user base, it's hard to find "best practices" for usage, so some of the ways we're using it are probably suboptimal.

The method we chose for object tracking - using colours - had numerous associated challenges. Finding the right kind of object to track was difficult: shiny things (like the red plastic lid we initially used and that can be seen in figure 9) are hard to work with because of how their apparent colour reacts to changes in angle, while dull colours are difficult to identify because they tend to blend in with the background. In the end, brightly coloured clothes were a suitable alternative.

Camera

Due to a difference in latency between the camera stream and the results of PTZ orientation queries, we had severe difficulties keeping the bomb images in the correct location. Because pan position data cames faster than video data, bombs would start to move before the image and destroy the illusion that they were a part of the image. To work around this, we noted that the delay was consistently around 700 milliseconds and simply delayed all updates of the camera's orientation by this amount. This kind of hard-coding would likely not be acceptable in a production system, where different kinds of network configurations and camera types might need to be treated in different ways. For our purposes though, it was sufficient (though certainly not perfect).

Miscellaneous

Some problems arose due to group members using different operating systems and development environments. Maven[27], in particular, caused problems for most members of the group at one time or other.

[27] https://maven.apache.org

We also had to compromise on our choice of programming languages. The original prototype of the object tracking functionality was written very quickly in C++, but we decided to port the code to Java instead since most members were much more comfortable working with Java than C++. This required finding a suitable Java wrapper for OpenCV (we investigated two different wrappers) and proved more time consuming than we had expected, but was regardless valuable for making the code more understandable for the whole group.

Alternative Approaches

In most areas of the project, we could have used different approaches. Here we will briefly discuss some of those approaches and the consequences they might have for the development process and final prototype.

Communication

Any distributed application needs to decide how communication between its constituent parts should be realised. One common way of solving this problem is through the use of a service-oriented architecture. Nowadays, this usually means exposing a set of services using the Web Services[28] protocol stack. While this approach is certainly a valid one, it does add to the complexity of an application as simple as ours by e.g. mandating the use of XML, imposing unnecessary security features, and requiring rather large chunks of software to handle the WSDL and SOAP languages in a meaningful way.

Other methods of communication were also considered, such as the use of websockets[29]. The WebSockets protocol is a, for it being a web protocol, fairly advanced protocol with low overhead. However, being advanced it is also a lot more complex than e.g. a REST-based approach. Purely TCP-based solutions were quickly dismissed as being too tedious to work with in a meaningful way.

As was described in the chapter on the web client, the client is tied to a single game server. During the development an alternative method was conceived, but never realised. It would make use of an Eddystone BLE beacon[30] to broadcast clients tied to specific game locations, rather than to game servers.

Object Tracking

Our chosen method for tracking the defusing player - based on colour - is quite limited in several ways. As noted previously, colour is very dependent on lighting conditions and angles, so we required players to wear brightly coloured clothing. If the game were to be extended to include more than one defuser, handling multiple colours would further complicate this problem. It might also be nice to be able to play the game without wearing a specific type of garment.

[28] https://www.w3.org/TR/ws-arch
[29] https://tools.ietf.org/html/rfc6455
[30] https://developers.google.com/beacons

An alternative approach could be to use Axis's Digital Autotracking API, which enables a PTZ camera to automatically follow a target. This is however problematic, since our game requires that one player is able to manually control the camera. A better alternative might be to use a more advanced computer vision method that can identify people in images instead of simply coloured objects, such as for example the skeletal tracking used in Microsoft's Kinect[31]. By fulling tracking a player's skeleton, we could implement more advanced functionality such as pointing at a bomb to diffuse it or some other kind of gesture-based control. These more advanced techniques are however outside of the scope of this course and would likely have required significantly more time and effort from us to implement. It's our view that the object tracking technique we used was sufficient for a prototype, but should be improved if development were to continue.

Another consideration is whether it might have been more appropriate to perform the image processing on the camera (in an ACAP, for example) rather than in the server application. If we did not also need to show the unprocessed video stream as part of the game, it would likely be better to do the processing on the camera in order to reduce the amount of data transferred over the network (though this probably isn't a huge concern). However, since we do need to show the video images, there is no real advantage to performing the object tracking on the camera directly (and there might be disadvantages in terms of the relative memory limitations and performance of the server and camera hardware).

Programming Languages/Frameworks

We decided to compromise on our choice of programming languages when working on the project. Since OpenCV is written in C++, it would have been practical to write that part of the application in C++, but this would have made things difficult if other parts were written in Java. Since most of the group is more comfortable with Java, we decided that the entire server application would be written in Java instead. Porting our finished object tracking code from C++ to Java proved more difficult than anticipated and wasted quite a bit of time - it might have been best to find a different solution, such as sticking with C++ or using some kind of framework to simplify C++ game development.

There exist numerous game development frameworks for C++, C# and Java that could have simplified our development process. We opted to develop the game directly in Java, but in hindsight the graphical elements of the game would probably have been much easier to implement with help from some kind of framework. Our choices here can mainly be attributed to a general lack of experience of Java game development in the group. In a longer project, we would have more time to investigate the available options before starting implementation. A pre-made game engine might also be appropriate, but can often have a steep learning curve.

[31] http://www.xbox.com/en-US/xbox-one/accessories/kinect

Conclusions and Further Work

While the game supports more than two players to some extent, it has not been a focus of development since it was not a requirement when we started. Consequently, a logical extension to the game would be to allow multiple players to defuse bombs simultaneously. It would then be necessary to add identification of multiple players via the camera and to keep track of team points, and remembering high score of the players and teams.

Adding puzzle elements when defusing bombs would add excitement to the game. The game could also be made to feel more “game-like” by the use of more feedback, for example if the phone vibrates with a loud bomb noise and the game displays a striking animation, it would feel more realistic if one fails to defuse a bomb. One could use the phone's flash to further create an effect of an exploding bomb. One could add a user interface for the audience which shows the score and statistics. Taking a picture or short video sequence of the terrified players would make it more enjoyable for the audience. Security and privacy are missing at the moment and it should be added to the game, so it will not leak sensitive information or make it possible to change information and cheating.

From a technical perspective, the game could be modified so different types of cameras are supported instead of only the Axis cameras with PTZ functionality. It would also be interesting to experiment with the affordances provided by different methods of person-tracking, as discussed above in the challenges section.

In conclusion, we have successfully developed the Asymmetric Augmented Reality Game. The project shows that it is possible to use a camera with PTZ functionality in a new, relatively unexplored, area. We have learned more about web services, connected systems, devices, image processing, and teamwork.