UNIVERSITATEA “ALEXANDRU-IOAN CUZA” IAȘI

FACULTATEA DE INFORMATICĂ

GRADUATION THESIS

Global-Offensive Artificial Bot Entity (GABE)

proposed by:

Tudor-Ștefan Păuleț

Session: July, 2024

Paper coordinator:

Lect. Dr. Alex-Mihai Moruz

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" IAȘI

FACULTATEA DE INFORMATICĂ

Global-Offensive Artificial Bot Entity (GABE)

Tudor-Ștefan Păuleț

Session: July, 2024

Paper coordinator:

Lect. Dr. Alex-Mihai Moruz

Contents

1 Introduction

Released in 2012, Counter-Strike: Global Offensive is a multiplayer first-person shooter game in which two teams, the Terrorists and the Counter-Terrorists, compete against each other to achieve different objectives. Although the game offers multiple game modes, this application targeted the standard bomb-defusal game mode.

A round of a Counter-Strike match has one symmetric objective and one asymmetric one. Achieving any objective will result in a team victory. The common objective is eliminating the enemy team. The team-dependent goal is determined by the game mode. In the standard game mode mentioned above, the Counter-Terrorists have to prevent the Terrorists from planting and detonating the bomb in one of the predefined sites.

Since the game's launch, bots have always been available in all game modes to provide balance to the two teams; however, their behaviour has always been less than optimal. Because of this, the only beneficial use of bots was acting as a controllable entity - dead players can possess a bot that is alive and on their team. In 2021, the bots were removed from the competitive mode of Counter-Strike, as the aforementioned facility was abused by players. In September 2023, Valve, the game developer, updated the engine behind the game from Source to Source 2 and renamed the game Counter-Strike 2, but the decision to not include the bots in the competitive mode was maintained. Bots are still present in the casual game modes and the practice environment.

Counter-Strike is one of the most popular multiplayer games and also one with the highest prize money for competitive tournaments, amassing 15.11 million in 2023. Because of this, Valve invested a significant amount of time and resources in anti-cheat software (VAC[17]) and making their game as inaccessible to interference from external sources as possible. For this reason, very few APIs providing game information exist. The lack of methods to access and analyse in-game information from an external environment will prove to be a challenge in the task ahead.

Given my passion for the game pre-2023 and considerable knowledge of the game mechanics, this application aims to be a rudimentary bot that is able to compete against the bots provided by Valve in a restricted environment of 1vs1[18] in the standard bomb-defusal mode. Moreover, the activity of playing a game is a task that involves a lot of decision-making based on multiple factors and taking actions in parallel with a tight synchronisation between them to achieve the goal. Given the challenging nature of Counter-Strike, but also the fact that this difficulty comes not from the numerous mechanics of the game

but rather from the precision that top-level players have in these simple actions makes this project an interesting application.

2 Similar applications

The starting point for my application was the bots provided by Valve. Although the implementation is proprietary, the in-game behaviour provides the following key points:

- the bots are aware of the match state and attempt to improvise and adapt to the needs of the team; (e.g. the bots tend to stick together while defending sites)
- the bots are aware of the game state, actively contributing towards the current team objective; (e.g. if the bot is in the Counter-Terrorist and the bomb is planted, this will drive them towards the planting point to defuse the bomb)
- players are able to communicate with the bots through a number of predetermined commands; (e.g. pressing **Z** + **4** yields the command "Hold this position" to ask the bot to hold the current site)
- bots have difficulty levels that changed the accuracy of shooting and reaction speed; this is the only aspect of their behaviour that is influenced by difficulty
- the aim of the bots is based on the in-game information received from the engine; (e.g. they can follow a live target with perfect precision)

In a 2021 paper, Cheyang Dai [1] described an agent based on two deep neural networks that was able to play the game. The aim approach used by the author is based on a convolutional neural network (YOLO model [2]) and the success of the implementation encouraged me to proceed with this idea of using a deep neural network to recognise enemies. The movement mechanism, on the other hand, is based on a custom-made neural network. This approach has one considerable advantage: the ability of the bot to learn different movements by analysing the behaviour of real players under the same circumstances. With enough experience, the bot can even identify a clear solution to obstacles and unique transition locations. On the other hand, I consider that the use of such a general approach misses one key mechanic of the game: the player's movement in a match heavily depends on the current map. Map knowledge is crucial to an above-than-average player, and for this reason a bot that attempts to mimic human performances should be aware of more than just the immediate environment and have an overall understanding of the game-state and the possibilities available.

3 Implementation details

3.0 Technical details

Almost all of the implementation of the approach described below is in C++, more specifically C++20, with only a single module written in Python. Due to the low-level dependencies (the window management system, see section 3.3), the target OS is Unix-based (Arch Linux), with X11 being the required displayed server. The Counter-Strike application runs natively on the target OS. Only one external dependency exists - CDS [8] - used in parsing JSON files.

Two testing environments were used: firstly, a desktop with 32 GB of RAM, a CPU with 12 cores and a Nvidia GTX 1050 TI graphics card; and secondly on a laptop with 16 GB of RAM, a CPU with 16 cores and a Nvidia RTX 3050 TI graphics card.

3.1 Task analysis

At first, a thorough analysis of the task at hand is required. Since the target behaviour is emulating a human player, the following must be achieved:

1. a precise aim, yet with a reaction speed that allows the opponent to react to your presence. Given the superior processing capabilities of an automatic bot compared to a human, it would be possible to create a bot that would kill any enemy as soon as it comes into the field of view.
2. movement through the environment with awareness of obstacles and means of avoiding them. A standard Counter-Strike map provides the player with obstacles that either require going through a particular spot to pass through or executing a different kind of movement to overcome (crouching, jumping).
3. aiming and anticipating possible enemy spots. In addition to recognising enemies, an essential aspect of the gameplay is predicting spots where the enemy may camp[9]. A player accustomed to the played map usually knows about these spots and can adapt his aim accordingly. The chances of being eliminated by an enemy increase when these spots are ignored.
4. team oriented behaviour. As previously stated, the Terrorists and the Counter-Terrorists have a map dependent objective, in addition to the objective of eliminating one another. Considering the importance of this objective, a bot playing the game should always keep this as one of its goals and attempt to complete it.

Given the list of requirements, we will introduce the following constraints for achieving a solution corresponding to them:

I. the bot shall play on only one map (de_dust2) in the bomb-defusal scenario - Considering the many aspects that depend on the chosen map, limiting to one is a reasonable choice for the success of the implementation. On the other hand, the implementation shall be modular and adapting to a new map should be done solely through updates to the map representation.

II. the bot shall first learn to play for the Terrorist side - Given the more active role of a Terrorist, starting with this as an objective should yield better results, as the playstyle of a Counter-Terrorist depends more on the enemy's behaviour that cannot be controlled.

III. shooting shall be the priority of the bot - Considering that one of the two objectives of a Counter-Strike player is eliminating your enemy, the bot shall concentrate on shooting

3.2 Environment analysis

The second step in developing the application was the environment analysis. For the bot to interact with the environment, it first needs information about the state of the game, the position and the presence of enemies in the field of view. As previously stated, Counter-Strike is strongly isolated from any external application trying to read or write any of the program's memory. Searching for an API that provides information regarding all the aspects that the bot needs to be aware of yielded only a single result: the Counter-Strike: Global Offensive Game State Integration (GSI) [3][4].

Game State Integration is an API created by Valve to provide third-party applications with real-time access to game information. This API is mainly used in major Counter-Strike tournaments to display player statistics and create real-time visual effects synchronised with important events of the match (bomb planting, bomb defusal, aces, clutches) as can be seen in image 3.2.1. A thorough analysis of Game State Integration can be found in section 3.3.

Although GSI does provide considerable information, aspects such as the player position, the current map zone, and the presence of enemies still need to be acquired. The second source of information is a

straightforward one: the user interface (UI). An example of the standard Counter-Strike interface can be seen in image 3.2.2.

image 3.3.1 GameStateIntegration being used to display player information

The top-left corner contains a mini-map, with the main sites being marked, and below it the current zone is displayed in white font. The bottom side of the screen holds information regarding the player: inventory in the right side, the health, armour, ammo in the centre and the money to the left. And of course, the presence of an enemy becomes apparent as soon as it comes into the field of view. Considering this large amount of information that the game provides through the UI, other sources of information might seem redundant. Although image analysis is an easy task for a human, the task of doing it programmatically is much more challenging, as we shall see in section 3.3

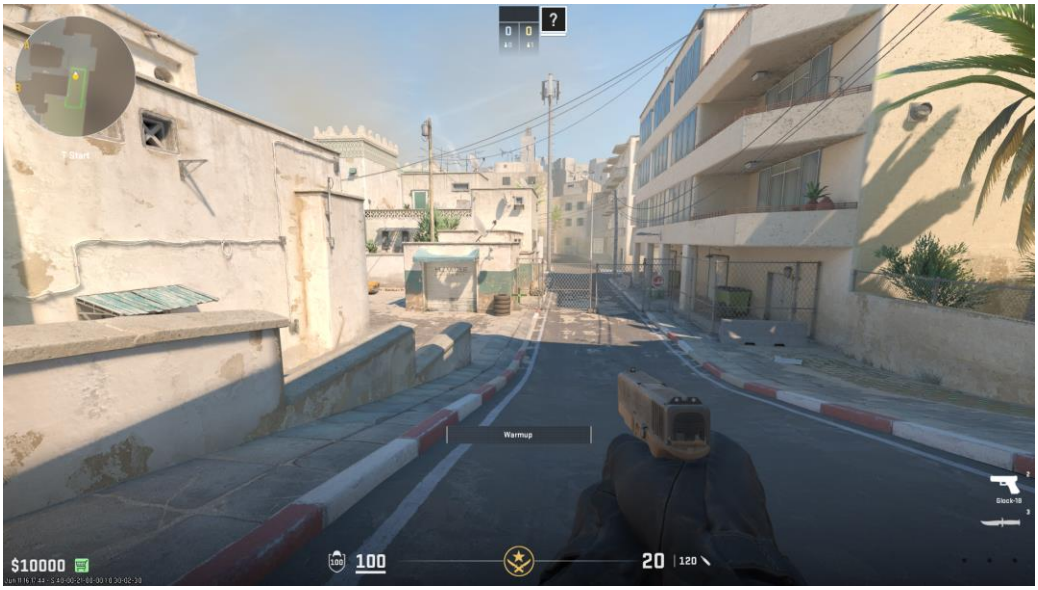


image 3.3.12 the standard Counter-Strike UI

The third and last source of information is the developer console. Most studios provide a way for users to interact directly with the engine that powers the game to enable them to develop additional tools and mods or debug game crashes. Valve does provide such a mechanism through the developer console (image 3.2.3).

The developer console is a mechanism that must be enabled in the game options. Afterwards, it can be accessed through the '~' key (or any other if this key bind is replaced in the options). This will create a pop-up window that allows the user to input predefined commands [5]. The developer console is a powerful mechanism and a crucial building block of this application, as we shall see later in this paper.

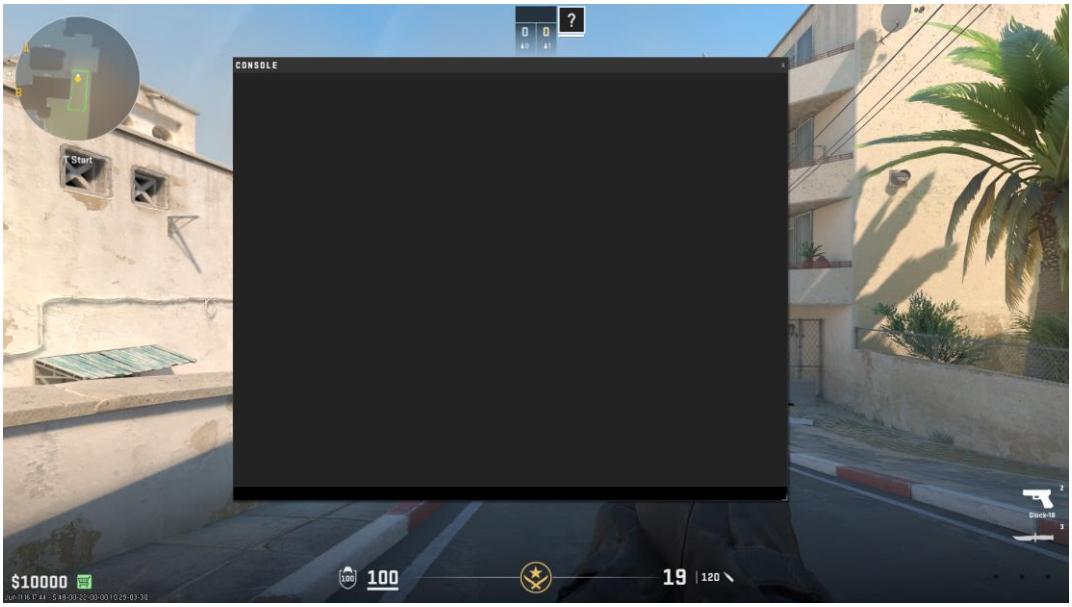


image 3.2.3 the Counter-Strike developer console

The last environment entity that provides information is the map itself. In image 3.2.4, we can see a general overview of de_dust2. Considering its structure, entry points to different zones, obstacles, and camping spots is crucial for a bot that aims to achieve points 2 and 3 discussed in section 3.1. Unfortunately, there are only two sources of information available: the UI that was previously discussed and my knowledge gathered through playing the game on this map. As we shall see in the following sections, the first source is unreliable, as it does not provide all the necessary information. Therefore, map navigation will depend on the experience of other users, but mainly on the personal one.

As observed, the map is divided in 3 major sides: the B side, on the left side of the map, the mid side, starting from above T spawn, and the A side, on the right side of the map. A further breakdown of the map will be needed in smaller regions, as demonstrated in section 3.3.

image 3.2.4 overview of de_dust2

3.3 Application components

We have seen that the bot needs to be aware of a considerable amount of information and that there are multiple sources of information. To ensure that the design of the application follows the SOLID principles [7], each source of information will be treated in a separate class, and an aggregator class shall be responsible for providing inputs to the components and handling the outputs. With that in mind, the following subsections will describe in detail the development process of each of these components.

3.3.1 Enemy detection

The main mechanic of Counter-Strike is shooting. A player's aim represents an important part of his gameplay. For this reason, the aim of the bot was regarded as a very important mechanic, if not the principal one. Because of this, an important amount of time was invested in developing this component and optimising it to ensure the best performance.

As stated in section 3.2, the only source of information regarding enemy presence is the UI. The first idea was to use an object detection algorithm based on a neural network to recognise enemies present in the field of view by using live images from the game captured with the C++ application.

Having no prior knowledge of image recognition or object detection, a thorough look through internet resources seemed like the only feasible option. Thanks to Ramona Albert, the process went much smoother, as she directed me towards the YOLO [9] algorithm and helped me with the appropriate resources.

Before diving into YOLO's particularities, the task of object recognition must be understood. In a normal classification problem on images, the task is simple: given any image that has only one central object and a set of possible labels, the objective is to assign one of the labels to the image. The problem becomes considerably more difficult when the image focuses on more than one object that may belong to different classes. In addition to simply labelling these objects, the task now also consists of identifying the image regions containing the objects. This spatial delimitation of objects is done through bounding boxes. Bounding boxes are tuples of 4 values *(x1, y1, w, h)* that have the following meaning: *w* and *h* represent the width and height of the bounding box, with *x1* and *y1* being the coordinates of the centre of the box.

Having made this digression, we can now return to the problem at hand: as stated, we only wish to detect enemies. Only one classification label will exist, the Counter-Terrorist class label. The bounding box problem, on the other hand, is more challenging. The bot's accuracy depends entirely on the accuracy of the bounding box prediction. A state-of-the-art algorithm for object detection is needed to ensure the quality of requirement 3.1.1. Not only that, but our detections must also be fast. In a game such as Counter-Strike, where the reaction speed makes the difference between eliminating the enemy or finding yourself eliminated by the enemy, our bot needs to have a reaction speed comparable to the average reaction speed of human players (that is about 150ms).

These requirements make YOLO the perfect candidate for the enemy detection task. The algorithm is based on a convolutional neural network (CNN) that is responsible both for predicting the bounding boxes, and the labels of the identified objects. Convolutional neural networks represent the modern basis of object detection as they capture a vital feature of human sight: proximity. Regular neural networks analyse pixels independently of one another. Opposite to this, CNNs consider a pixel's neighbouring values as part of its importance. This change allows CNNs to provide remarkable results in image recognition.

The backbone of convolutional neural networks is convolutional layers. Convolutional layers make use of the mathematical concept of a convolution [12] and do so by using kernels, a kernel being a square matrix of small dimensions (usually 3 x 3, 5 x 5 or 7 x 7). Convolving a kernel over an image is directly linked to the proximity mentioned above, as it takes into consideration multiple neighbours in addition to the currently analysed pixel. By using this mechanism, CNNs are able to consider regions of the image instead of individual pixels.

In contrast to fully connected layers that are defined using only one parameter, the size, two parameters define convolutional layers: the size of the kernel on the respective layer and the number of kernels present. Given that kernels are square matrices, applying convolutions with multiple kernels over an image produces multiple matrices. This results in the data having a 3-dimensional shape while flowing through the network.

Due to the significant size of images (e.g**.** a 640 x 640 RGB image is represented using 1.228.800 integer values), convolutional neural networks have two mechanisms to reduce dimensionality:

- instead of considering convolutions over the entire image, a stride is considered for sliding the convolutions by more than one step, leading to some rows and columns being skipped. Doing so produces outputs of lower height and width.
- altered with convolutional layers, pooling layers exist that have a function similar to the convolutions: given a 3-dimensional input, pooling layers squash regions of the image based on a selected policy and produce a 3-dimensional output of the same depth but reduced width and height. For example, a 2x2 max-pooling layer processes the matrix in such a way that each 2x2 region produces a unique element, in this case the maximum out of the 4.

YOLO is a one-shot object detection algorithm. This means that, by only running the image one time through the neural network, it outputs the bounding box predictions and labels. YOLO is also designed such that it can detect multiple objects present in the same image: the output is an array of *(bounding-box, label)* pairs. The algorithm has gone through multiple stages of development, paper [2] describing the first version, YOLO v1. In order to keep things simple and considering my limited knowledge in the machine learning domain, this was the targeted version.

The intention of the thesis is not to reuse the existing implementation. Therefore, the initial approach was to implement the algorithm from scratch utilising the original YOLO v1 paper [2]. Since this is a crucial building block in terms of speed, this implementation was written in C++.

To obtain the said implementation, several prerequisites are needed:

- the ability to build a CNN with layers of custom sizes and activation functions.
- the capacity of propagating forward an image and receiving the prediction in the shape of an array.
- the possibility to backpropagate a prelabeled dataset in order to have a model that learns from the dataset and optimises its weights.

Taking inspiration from Python's Numpy library, the starting point in the development of the machine learning library would be data structures required in mathematical calculus, first of which was the **LinearArray** class.

One requirement was clear from the beginning: the ability to catch as many errors as possible at compile time. In opposition to the way Python handles issues related to operations between numpy arrays of mismatching sizes, array resize into an invalid size and so on, the design of **LinearArray** allows identifying such issues at compile time.

In order to keep the flexibility of a class capable of performing mathematical operations on vectors of any dimension, yet still handle any problems at compile time, the building blocks of the **LinearArray** class were C++ Templates [9][10].

C++ Templates are "entities parameterized by one or more template parameters" that the compiler expands fully at compile-time, meaning that all the mentioned template parameters must be fully defined at compilation. C++11 introduced variadic templates that allow an unspecified number of parameters (usually limited to 1024) to be passed as template parameters, a feature which will be used extensively to customise the needed classes.

Another crucial feature of C++ Templates is template specialisation: given a templated class, a part (partial specialisation) or all the parameters (full specialisation) may be specified in a new definition of the class template, allowing for an entirely new definition of the respective class. These two features combined allow for a powerful mechanism of creating self-recursive class definitions, the fundamental mechanism of **LinearArray**. This idea of self-recursion comes from the following remark: a 1-dimensional vector is a collection of values of a given type. A 2-dimensional vector is a vector of 1-dimensional vectors. A 3-dimensional vector is a vector of 3-dimensional vectors. And so on. Now, we can see the reasoning behind defining a **LinearArray** of n dimensions in terms of a **LinearArray** of n-1 dimensions.

At first, the self-recursive definition of **LinearArray** may seem replaceable with a typical dynamic vector of vectors. However, using templates provides several advantages:

1. template coding encourages inlining - Inlining is an important feature of the C++ optimizer, when function calls are replaced with the body of the function and a lot of overhead is reduced in this manner, thus the performance is increased. Moreover, performance is further improved as the compiler has more knowledge about the code when parameters are constrained through templates.
2. Determinism - Usage of templates instead of unconstrained self-inheritance provides an additional layer of safety. Defining a class recursively poses the danger of infinite recursion. Using template expansion ensures that in such a case, the compiler shall reach the template expansion limit and signal an error at compile time. Not only is this good because the error is caught earlier than at runtime, but it is also more explicit and easier to treat.
3. compile-time assertions - As stated previously, all template parameters must be specified at compile time. This also allows us to make use of *static_assert* assertions that are fully resolved at compile-time. Through them, out of bounds accesses, operations with LinearArrays of non-matching sizes or mathematical operations impossible on vectors of incompatible sizes can all be prevented at compile time.

Having motivated the design choices, let's dive into the actual implementation. In image 3.3.1.1 we can see the definition of the generic **LinearArray**; it has a template type parameter, *DataType,* that will represent the standard data type of the elements contained, a required non-type parameter *first_size* that prohibits **LinearArrays** of size 0 from existing and the last parameter *remaining_sizes* that is a variadic non-type parameter that allows for the self-recursion to happen. As can be seen, the class does not directly inherit itself, but the recursion happens, as we will see, through the LinearArrayContainer class.

```
template <typename DataType, Size first_size, Size... remaining_sizes> class LinearArray :
    public linearArray::impl::LinearArrayContainer<LinearArray<DataType, remaining_sizes...>, first_size>,
    public linearArray::impl::LinearArrayGenericOps<LinearArray<DataType, first_size, remaining_sizes...>>
```

image 3.3.1.1 the generic LinearArray class definition

In image 3.3.1.2 we have the definition of the LinearArrayContainer class. It has two template parameters: the type template parameter *DataType* and the single non-type template parameters *s*. An important thing to remark here is that the *DataType* from this class is not the same as the one passed to the LinearArray class; the *DataType* parameter in LinearArrayContainer is another LinearArray (line two

from image 3.3.1.1). LinearArrayContainer has a member *_data* that contains *s* elements of type *DataType*. The recursion happens here and the relation between a n-dimensional vector and a (n-1)-dimensional vector can be clearly seen.

```
template <typename DataType, Size s> class LinearArrayContainer {
  static_assert(s != 0, "Size of linear array must not be 0");

public:
  LinearArrayContainer() = default;
  LinearArrayContainer(LinearArrayContainer const&) = default;
  LinearArrayContainer(LinearArrayContainer&&) noexcept = default;
  explicit LinearArrayContainer(std::array<DataType, s> const& data) : _data(data) {}
  auto operator=(LinearArrayContainer const&) -> LinearArrayContainer& = default;

  constexpr auto& data() { return _data; }
  constexpr auto const& data() const { return _data; }
  static constexpr auto size() { return s; }

private:
  std::array<DataType, s> _data;
};
```

image 3.3.1.2 the **LinearArrayContainer** class definition

The LinearArrayGenericOps represents a collection of operations invariant to the array's dimensionality: addition, subtraction, pairwise multiplication, applying transformations, reductions and iterating. By separating the data containment logic from the implementation of operations, we respect the Single Responsibility and Dependency Inversion principles.

As with any recursion, the LinearArray class must also have at least one base case: in our case, there are two base cases:

- The first is the 1-dimensional array (image 3.3.1.3): the base non-recursive case. It has no dependency on another **LinearArray** instance of lower dimensions (as mentioned, 0-dimensional arrays would be ill-formed) and does not have any particular methods, therefore using the generic operations provided by default.

```
template <typename DataType, Size size> class LinearArray<DataType, size> :
    public linearArray::impl::LinearArrayContainer<DataType, size>,
    public linearArray::impl::LinearArrayGenericOps<LinearArray<DataType, size>>
```

image 3.3.1.3 the base case **LinearArray**

- The second is the 2-dimensional array, representing the concept of matrix. Given the particular operations available (flipping, convolutions, transposing, matrix product, etc.) this base case was required to provide all these features. Unlike the library used for inspiration, NumPy, that allows the usage of the GPU for accelerated computation, my implementation is CPU-bound due to complexity reasons. To minimise the speed-gap, multithreading is utilised in time consuming operations, such as matrix multiplication.

Another useful feature of **LinearArray** is its scalable serializability. Having the LinearArray as the underlying data structure for neural networks, serialising the network in a file after training comes down to serialising a group of **LinearArrays**.

Besides the benefits of using templates described earlier in this paper, the chosen implementation provides additional such advantages. As seen in the **LinearArrayContainer** class, the underlying container for the **LinearArray** class is *std::array*. This type of array is required to use contiguous stack memory. Such a memory allocation scheme ensures a higher degree of performance, as the entire memory layout of the data structure is known. Additionally, the contiguous memory allocation scheme mentioned enables the CPU to make full usage of cache locality.

The second step in implementing YOLO was the ability to build a standard neural network, a precursor to the convolutional neural network required for object detection. Simple neural networks, as well as convolutional ones, depend on one key aspect: the relationship between adjacent layers. Every layer must be aware of its successor to be able to feed information during forward propagation and also aware of its predecessor to send back gradients during back propagation.

Every layer is characterised by two components: the size, which gives the number of neurons, and the activation function. The last layer is unique, as it also has a cost function associated with it. A remark worth making is that the size of a layer by itself holds little meaning, as the weights of the network are related to the sizes of two adjacent layers. Considering all of these aspects, the building block of our neural network is the **LayerPair** class (image 3.3.1.4a) that aggregates instances of the **Layer** class (image 3.3.1.4b).

```
template <typename DataType, typename FirstLayer, typename SecondLayer, typename... RemainingLayers> class LayerPair :
    public LayerPairContainer<DataType, NDLType<FirstLayer>::template Type<DataType>::dimension,
                              NDLType<SecondLayer>::template Type<DataType>::dimension,
                              LayerPair<DataType, FirstLayer, SecondLayer, RemainingLayers...>>,
    public LayerPair<DataType, SecondLayer, RemainingLayers...> {
```

image 3.3.1.4a the **LayerPair** class

```
template <typename DataType, typename ActivationFunction, typename InitializationScheme, typename Dim> class Layer :
    private ActivationFunction {
```

image 3.3.1.4b the **Layer** class

The **LayerPair** class is based on the same idiom of template expansion. As seen, the **LayerPairContainer** depends on two adjacent layers, therefore the dimension of the weights, which are 2-dimensional **LinearArrays**, can be computed.

As mentioned previously, serialisation is crucial in a neural network. As the entirety of the network can be obtained from the weights, serialisation of the network represents serialisation of the weights, resulting in the **LayerPairContainer** being responsible for this aspect.

The base case of the recursion is more straightforward in this case: the last layer pair receives as a parameter a special type of layer that has associated a cost function in addition to the usual parameters.

The generic feedforward and backpropagation functions can be seen in image 3.3.1.5. In the final layer, they are specialised so that feedforward returns the final output of the entire network, while backpropagation applies the cost function to compute the corresponding gradients.

```
auto feedForward(Input const& input) {
  return NextLayerPair::feedForward(SecondLayerType().feedForward(input, weights()));
}

template <typename Target, typename Clipper = gabe::utils::math::IdentityFunction<>>
auto backPropagate(Input const& input, Target const& target, DataType learningRate, Clipper&& clipper : Clipper
  auto nextLayerGradient =
      NextLayerPair::backPropagate(SecondLayerType().feedForward(input, weights()), target, learningRate, cl
  auto [kernelGradient, currentLayerGradient] = SecondLayerType().backPropagate(input, weights(), nextLayerG
  kernelGradient.transform(clipper);

  weights() -= kernelGradient * learningRate;
  return currentLayerGradient;
}
```

image 3.3.1.5 feedforward and backpropagation

As can be seen, backpropagation is parametrized with two template parameters: the *Target* and the *Clipper* parameter. The necessity of the *Target* parameter is motivated by the fact that, although the size of the last layer in our network corresponds to the size of the target vector, layers are only aware of the

sizes of adjacent layers and cannot know the network's final output. By using the *Target* parameter, we can check for a mismatch between the target vector and the final layer output at compile time, yet maintain the property that each layer only contains information about adjacent layers. The *Clipper* parameter was introduced in the later stages of development while trying to fit a more complex model, not a standard neural network. It allows the network to provide a gradient-clipping function to prevent gradient values from being too large.

After completing the implementation of a standard neural network, the last aspect to treat is adding convolutional layers to a neural network. Convolutional layers come with more parameters than standard layers, as can be seen in the class definition in image 3.3.1.6

```
template <typename DataType, typename Input, typename DepthDim, typename KernelDim, typename ConvolutionFunction,
          typename ActivationFunction, typename InitializationScheme>
class ConvolutionalLayer : private ActivationFunction, private ConvolutionFunction {
```

image 3.3.1.6 the **ConvolutionalLayer** class

The feedforward and backpropagation implementations are no longer as straightforward as in the standard Layer case, and a thorough documentation of all mathematical computations required can be found in [11].

Although the **LayerPair** implementation of the feedforward and the backpropagation functions is invariant to the underlying **Layer** implementation, two problems arise: the size of the weights when a convolutional layer is part of a pair and the transition from the last convolutional layer to the first fully connected one. This process is known as flattening, as the 3-dimensional shape of the data is compacted into a 1-dimensional vector for further processing in the fully connected layers. The responsibility of flattening the 3D **LinearArray** between the last convolutional layer and the first fully connected one falls on the **LayerPair** class. To treat this problem in a uniform manner, corresponding to the **LayerPair** implementation that already existed, a simple specialisation of the **LayerPair** class was required, as can be seen in image 3.3.1.7

This class has two responsibilities: to flatten the input as it goes through the networks, as can be seen in the *feedforward* step, and to propagate backwards a compatible **LinearArray**, to conform to the type expected in the previous layer.

```
template <typename B, typename D, typename DataType, typename FirstLayer, typename SecondLayer>
class ConvolutionalLayerPairFlattener {
  using Input = typename FirstLayer::template OutputType<DataType>;

public:
  auto feedForward(Input const& input) {
    typename D::Input flattenedInput {input.flatten()};
    return static_cast<D*>(static_cast<B*>(this))->feedForward(flattenedInput);
  }

  template <typename Target, typename Clipper = gabe::utils::math::IdentityFunction<>>
  auto backPropagate(Input const& input, Target const& target, DataType learningRate, Clipper&& clipper : Clipper  = Clippe
    typename D::Input flattenedInput {input.flatten()};
    auto rez = static_cast<D*>(static_cast<B*>(this))->backPropagate(flattenedInput, target, learningRate, clipper);
    return Input {rez.flatten()};
  }
};
```

image 3.3.1.7 the **LayerPair** flattener specialisation

The second issue with the network's weights when using convolutional layers is that, in the case of fully connected layers, the sizes of adjacent layers dictate the size of the weights between them; when convolutional layers are considered, the situation is actually reversed. The size of the kernels and the number of kernels in a layer represent the actual weights between that layer and its predecessor, while the relationship between the dimensions of the previous layer combined with the current layer kernels gives the shape of the output, namely the layer size.

To better understand this, let's take an example: assume we have two fully connected adjacent layers: layer A of size 10 and layer B of size 8. Between layers A and B exists a matrix of weights of dimension 8 * 10 = 80 individual weights. The output of these layers is an array of size 8 (or rather 8x1 as we work with 2-dimensional arrays at all times in a simple neural network).

In the second example we have two convolutional layers: layer C that outputs a 10x10x3 image (10x10 image over 3 channels) and layer D that has 5 kernels of size 3x3 (the depth is implicitly equal to 3, the depth of the output from the previous layer). The number of weights is equal to the number of weights in the kernel, meaning 5 * 3 * 3 * 3 = 135 individual weights. Given a stride of 1 for the convolution, the output of the D layer would be of size 10x10x5 (10x10 as the image size does not change, 5 as the number of kernels involved in the D layer).

Convolutional neural networks introduce a second type of layers: pooling layers (image 3.3.1.8). They have a considerable number of parameters, just like convolutional layers, and they conform to all the properties specified above, with one exception: pooling layers produce no weights. As the pooling layers

act directly on the input, there is no need for a **LayerPair** between a convolutional layer and a pooling layer to produce any weights.

```
template <typename DataType, typename Input, typename Dim, typename StrideDim, typename PoolingFunction>
class PoolingLayer : private PoolingFunction {
```

image 3.3.1.8 the **PoolingLayer** class

Before attempting to train the neural network, I added one last feature to the layers: the **InitializationScheme**. At first, I was using random initialization from the [-1, 1] interval to initialise the weights in all the layers. This way of initialization may be effective in the case of simple neural networks with fully connected layers. For CNNs, this is no longer the case. Special initialization schemes, such as the He or Xavier [16] initialization schemes are needed to ensure faster network convergence. Considering that each layer may need a different initialization, incorporating the possibility to pass a custom initialization scheme to each layer was required.

After developing all the necessary tools to create a neural network, the last remaining aspect was to aggregate them into a class that could easily handle a labelled dataset and back propagate it through the layers: the **NeuralNetwork** class (image 3.3.1.9).

```
template <typename DataType, typename InputLayer, typename... Layers> class NeuralNetwork :
    public impl::LayerPair<DataType, InputLayer, Layers...> {
```

image 3.3.1.9 the **NeuralNetwork** class

As can be seen in the definition, a **NeuralNetwork** is simply a wrapper around a **LayerPair** that encompasses all the layers. The main feature of this class is found in the backpropagation methods (image 3.3.1.10).

The second method uses serialisation during training the model with possible faults in mind. Should the code execution or the entire computing unit fail, progress is not entirely discarded. Another advantage of using serialisation while training is that should divergence of the model occur, the last stable model can be analysed to determine issues of the model.

LabelEncoding converts single-value labels into a form suitable for the network (e.g. one hot encoding).

```
template <typename Input, typename LabelEncoderType> auto backPropagate(Size epochCount, DataType learningRate,
                                                                         ImageDataSet<Input> const& dataSet,
                                                                         LabelEncoderType&& labelEncoder) -> void {
  for (Size idx = 0; idx < epochCount; ++idx) {
    for (auto const& e : dataSet.data()) {
      backPropagate(e.data, labelEncoder(e.label), learningRate);
    }
  }
}

template <typename Input, typename LabelEncoder>
auto backPropagateWithSerialization(Size epochCount, DataType learningRate, ImageDataSet<Input> const& dataSet,
                                    LabelEncoder&& labelEncoder, std::string const& serializationFile) -> void {
  for (Size idx = 0, propIdx = 0; idx < epochCount; ++idx) {
    for (auto const& e : dataSet.data()) {
      backPropagate(e.data, labelEncoder(e.label), learningRate);
      ++propIdx;
      if (propIdx % 20 == 0) {
        std::cout << "Serialization " << propIdx << " done\n";
        serialize(serializationFile);
      }
    }
  }
}
```

image 3.3.1.10 backpropagation in **NeuralNetwork**

With the tools necessary to build the YOLO model ready, the only aspect left to choose were the network's parameters. The YOLO v1 model made use of GPU computation so the entire model was too expensive to use. Considering the limitations of my implementation, I chose a network of more constrained dimensions that followed the same general directions (image 3.3.1.11).

```
using InitializedObjectRecongnitionNet =
    NeuralNetwork<double, ConvolutionalInputLayer<640, 3>,
                  FullStridedConvolutionalLayer<32, 7, 2, LeakyReluFunction, HeInitialization<>>, MaxPoolLayer<2, 2>,
                  FullStridedConvolutionalLayer<32, 5, 2, LeakyReluFunction, HeInitialization<>>, MaxPoolLayer<2, 2>,
                  FullConvolutionalLayer<64, 3, LeakyReluFunction, HeInitialization<>>, MaxPoolLayer<2, 2>,
                  FullConvolutionalLayer<64, 3, LeakyReluFunction, HeInitialization<>>, MaxPoolLayer<2, 2>,
                  FullConvolutionalLayer<128, 3, LeakyReluFunction, HeInitialization<>>, MaxPoolLayer<2, 2>,
                  FullConvolutionalLayer<256, 3, LeakyReluFunction, HeInitialization<>>,
                  InitSizedLayer<1024, Layer, HeInitialization<>, LeakyReluFunction>,
                  InitSizedLayer<outputSize, OutputLayer, XavierInitialization<>, SigmoidFunction, YoloCostFunction>>;
```

image 3.3.1.11 the final object-recognition convolutional neural network

The first two strided convolutional layers used large convolutions to capture relations in bigger regions of the images. The pooling layers were used to reduce dimensionality. The last four convolutional layers were used to further process relations between the adjacent pixels. Finally, two fully connected layers were used to predict the final labels. Initialization schemes were chosen according to the network's structure to ensure convergence.

With the network built and defined, training required labelled data. A YOLO based model needs images labelled with bounding boxes and class labels to train. Moreover, the amount of data should be fairly consistent; otherwise the model may fail to correctly extrapolate. Luckily, I managed to find a prelabeled data set [13] with over 1200 train images and 185 test images.

The data came in pairs of *(image.jpg, labels.txt)* and the *labels.txt* file consisted of multiple lines, each defining a valid detection for the corresponding image in the form *(class, xCenter, yCenter, width, height).*

Parsing the label files was straightforward, as their structure implies reading each line and converting it to a suitable representation for backpropagation. On the other hand, jpg images needed an external library in order to be read. For this purpose I used *libjpeg* [15], a library for processing jpg files (see image 3.3.1.12).

```
FILE* in = fopen(filePath.c_str(), "rb");

struct jpeg_decompress_struct cinfo;
struct jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_decompress(&cinfo);

jpeg_stdio_src(&cinfo, in);
jpeg_read_header(&cinfo, TRUE);
jpeg_start_decompress(&cinfo);

int width = cinfo.output_width;
int height = cinfo.output_height;
int numChannels = cinfo.num_components;
assert(width == R::InnerLinearArray::InnerLinearArray::size()
       && "Width of image should match the third dimension of the container");
assert(height == R::InnerLinearArray::size() && "Height of image should match the second dimension of the container");
assert(numChannels == R::size() && "The number of channels should match the first dimension of the container");

auto* imageBuffer = new unsigned char[width * height * numChannels];

while (cinfo.output_scanline < cinfo.output_height) {
  unsigned char* row_pointer = &imageBuffer[cinfo.output_scanline * width * numChannels];
  jpeg_read_scanlines(&cinfo, &row_pointer, 1);
}

jpeg_finish_decompress(&cinfo);
jpeg_destroy_decompress(&cinfo);
fclose(in);
```

image 3.3.1.12 decompressing a jpg image into an array of bytes

Having labelled and transformed the images into a valid representation for the network, the goal of training the model was the last remaining aspect. The implementation of the training process can be seen in image 3.3.1.13.

```
auto dataSet = loadCS2Images<LinearArray<double, 3, 640, 640>>("../data/yolo/cs2/train");
dataSet.normalize();
std::random_device rd;
std::mt19937 g(rd());
std::shuffle(dataSet.data().begin(), dataSet.data().end(), g);

ObjectDetection::InitializedObjectRecongnitionNet nn {};
nn.deserialize("deepObjectDetectionMarker.nn");
auto start = std::chrono::system_clock::now();
auto clipper = [](double value) {
  if (constexpr auto precision = 2.0; std::abs(value) > precision) {
    return value < .0 ? -precision : precision;
  }
  return value;
};
nn.yoloBackPropagateWithSerialization(30, 0.0001, dataSet, "deepObjectDetection.nn", clipper);
nn.serialize("deepObjectDetection.nn");
nn.yoloBackPropagateWithSerialization(75, 0.001, dataSet, "deepObjectDetection2.nn", clipper);
nn.serialize("deepObjectDetection2.nn");
nn.yoloBackPropagateWithSerialization(30, 0.0001, dataSet, "deepObjectDetection3.nn", clipper);
nn.serialize("deepObjectDetection3.nn");
nn.yoloBackPropagateWithSerialization(30, 0.00001, dataSet, "deepObjectDetection4.nn", clipper);
nn.serialize("deepObjectDetection4.nn");
```

image 3.3.1.13 training the model

The prediction speed of the model was encouraging, given the full CPU bound computation: 80ms/prediction, or about half of the average reaction speed of a Counter-Strike player. However, the training process posed two problems: the first, and minor one, was the size of the network and the second, critical one, was the divergence of the model.

As can be seen both in the dataset structure [13] and in image 3.3.1.11, the target images were of size 640 x 640 pixels in RGB format. Although multiple layers of max pooling were used to reduce the size of subsequent layers and, implicitly, the number of weights in the network, the final size of the model was quite large: ~3GB or approximately 800.000.000 weights. As mentioned in section 3.0, the desktop used for testing could easily fit this into memory. The problem concerned another aspect: the stack size. As mentioned previously in this section, the entire network relies on the **LinearArray** class, which in turn depends on the *std::array* class from the C++ standard; and that is a data-structure allocated entirely on the stack. Although this does improve performance, the standard stack size for a Linux executable (without modifications of these parameters prior to execution) is only 8MB. This obviously could not accommodate the memory requirements of the network and produced the *SIGSEGV* signal on each run due to a stack overflow. Without the required experience, this issue eluded me for a couple of days, which slowed down progress of the application. However, the issue was solved shortly after identifying the problem by increasing the stack size prior to calling the *train* method.

The second issue, the model's divergence, proved to be irreparable. Due to unknown reasons, the model failed to produce the expected results. The first issue was related to the weights starting to grow too fast (a problem known as exploding gradients). In order to prevent this from happening, the gradient clipping mechanism present earlier was introduced. Although this did indeed limit the magnitude of the weights, the model remained unstable, leading to the second issue: NaN (not a number) values. The model used single precision floating points for the weights, but after several iterations the weights in the network became undefined (NaN).

In the beginning, this only seemed to be a continuation of the first issue. However, analysis of the progress of the weights through iterations showed nothing of this: the weights seemed all stable in an iteration, but after propagating only one more training image, most of them became undefined. Having identified the problematic state of the network, I tried to capture a snapshot of it and investigate from there. The problem became evident once I tried to pass an example image through the snapshot: it only predicted boxes of size 0 and size equal to the entire image size. This was a disastrous result, as it showed that the model could not provide relevant information. The first idea to solve this was to reduce the learning rate and vary it through iterations; unfortunately, the result remained unchanged.

Searching for explanations behind this anomaly were also in vain, as no helpful source of information was found. Seeing that training the model for a certain period of time resulted in this issue, the second idea came to mind: using an intermediate state to see whether it managed to learn any sort of features. However, the number of iterations for which the model didn't diverge was far too low (about 1500 examples could be propagated before divergence), therefore this idea was also a failure.

Having already invested a considerable amount of time into debugging the network, I decided that it was for the best to abandon this momentarily, use an already trained model to detect enemies, develop the remaining aspects of the bot and return to the network after finishing all the components.

That brought me to the second alternative for object detection: the dataset that I used also provided a pretrained model for testing. However, in order to use this model, I needed to use Python with the Roboflow library. This need to use Python introduced a slight issue: as my implementation was in C++, I needed a way to communicate from within my program with a Python script, translate the predictions into a proper format and return them to my application.

The communication between the program and the script was realised through pipes (image 3.3.1.14). The C++ process forks and creates a child process. Before launching the Python script, the standard input and

standard output file descriptors are overwritten so that two pipes act as the communication channels between the child and its parent.

```
_childPid = fork();
if (_childPid == -1) {
  throw exceptions::ChildCreationException {};
}

if (_childPid == 0) {
  close(inPipe[0]);
  close(outPipe[1]);

  dup2(inPipe[1], STDOUT_FILENO);
  dup2(outPipe[0], STDIN_FILENO);

  execlp((scriptRootPath + "/.venv/bin/python3").c_str(), (scriptRootPath + "/.venv/bin/python3").c_str(),
         (scriptRootPath + "/objectDetection.py").c_str(), NULL);
} else {
  _channel[0] = inPipe[0];
  _channel[1] = outPipe[1];

  std::string checkMessage = "checking communication";
  write(_channel[1], (checkMessage + "\n").c_str(), checkMessage.length() + 1);

  auto* response = new char[64];
  read(_channel[0], response, 64);
  std::string responseString {response};
  delete[] response;
  if (responseString != "response:" + checkMessage) {
```

image 3.3.1.14 launching the Python object detection model as a child process

After overwriting the file descriptors, the script (image 3.3.1.15) is launched into execution. The parent would then pass an image as an array of bytes through the pipe and receive the corresponding bounding box predictions (image 3.3.1.16).

```
from roboflow import Roboflow
import supervision as sv


def read_image():
    byte_data = sys.stdin.buffer.read(640 * 640 * 3)
    image_array = np.frombuffer(byte_data, dtype=np.uint8)
    image_array = image_array.reshape((640, 640, 3))
    return image_array

data = sys.stdin.readline().strip()
response = "response:" + data
sys.stdout.write(response)
sys.stdout.flush()


rf = Roboflow(api_key="xL0E5GuDc339bLtw3p1A")
project = rf.workspace().project("csgo-2-hfa81")
model = project.version(1).model

sys.stdout.write("**finishedsetup**")
sys.stdout.flush()

while True:
    data = read_image()
    result = model.predict(data, confidence=40, overlap=30).json()

    detections = sv.Detections.from_inference(result)
    sys.stdout.write(str(detections.xyxy))
```

image 3.3.1.15 the object detection Python script

```cpp
auto analyzeImage(unsigned char* data) -> std::vector<BoundingBox> {
  static constexpr auto imageSize = expectedScreenWidth * (expectedScreenHeight - screenHeightOffset) * 3;

  auto* response = new char[256];
  write(_channel[1], data, imageSize);
  read(_channel[0], response, 256);
  std::regex regex {R"(\[\d+(,| )*\d+(,| )*\d+(,| )*\d+\])"};
  std::string string {response};
  delete[] response;

  std::vector<BoundingBox> result {};
  for (std::smatch stringMatch; std::regex_search(string, stringMatch, regex);) {
    auto match : string  = stringMatch.str();
    std::stringstream stringstream {match.substr(1, match.size() - 1)};
    int x, y, z, t;
    char delim;
    stringstream >> x >> delim >> y >> delim >> z >> delim >> t;
    result.emplace_back(Point {x, y}, Point {z, t});
    string = stringMatch.suffix();
  }
  return result;
}
```

image 3.3.1.16 communication between the C++ program and the Python script through the pipes

As can be seen, the mechanism is quite simple. The C++ executable sends 640 * 640 * 3 bytes of data at a time and the script reads them from the pipe, interpreting it as a 640 x 640 RGB image, passes it through the model and writes the predictions in the pipe in the following format: [*prediction_1, prediction_2, … prediction_n]* where each prediction is in the following format: *[x y w h]*.

Even though the precision of the predictions was remarkable, the speed proved to be unacceptable: ~800ms/prediction, which is 5 times slower than the average player's reaction speed. This performance contradicted objective 1 from section 3.1, so the results still needed improvement.

As the model provided was from an external source, I had no control over the source code. Any effort in this direction would have no result. This setback guided me to the third, and final, variant of the object detection component: training a YOLO model with Ultralytics.

Having already analysed and benchmarked the performance of a YOLO based model, it became clear that using this kind of object detection algorithm was the right choice. My lack of experience in the Machine Learning domain was the main reason behind the failure of the first idea. The second idea helped me create a communication channel from the executable to a Python script, an idea that could be tailored around any other model, not necessarily the Roboflow one. With these in mind, I used the Ultralytics library to train a YOLOv8 mini model with the already collected data for predictions (image 3.3.1.17).

As the performance needed to be as high as possible, I also used CUDA to enhance computation speed through the usage of the GPU. The results proved to be well above expectations: a precision

```python
from ultralytics import YOLO
import torch

torch.cuda.empty_cache()

model = YOLO('yolov8n.pt')

results = model.train(
    data='data.yaml',
    imgsz=640,
    epochs=100,
    batch=16,
    name='yolov8n_mini_custom'
)
```

image 3.3.1.17 training a YOLOv8 model

equivalent, if not even better, than the Roboflow model and a speed of 20ms/predictions. The model's remarkable speed well compensated for the overhead introduced with the communication through the pipes.

The script structure in image 3.3.1.15 suffered only two modifications: first, the Roboflow model was replaced with a locally trained YOLO model, and second, the array of data was passed in RGB format; however, due to backwards compatibility reasons from CV2, the model was expecting BGR format; this was solved with a simple rearrangement of the bytes.

Having obtained a model that fits both the requirement for precision and speed, the enemy-detection component was finalised and development could be focused on the next feature.

3.3.2 Position reading

The second aspect of concern was gathering real-time information about the bot's position and orientation. Naturally, as a 3D shooter, the position and orientation are tuples of 3 values: *(x, y, z)*. Of concern are usually the *x* and *y* coordinates both for the position and the orientation. The *z* coordinate for the position may interest us under certain circumstances, as the position may vary based on elevation. However, the *z* component of the angle is of no use, as Counter-Strike has no mechanism of rolling the camera in that axis.

As stated in section 3.2, no standard source of information from the game provides any of the required information. However, the developer console mentioned in section 3.1 proves to be the solution to this issue. One of the commands available in the developer console is the "cl_showpos 1*"* command. As can be anticipated from the syntax, this command sets a flag of the game engine to the "1*"* value, which in turn prompts it to display information about the player's position, orientation, and velocity in the UI (image 3.3.2.1).

image 3.3.2.1 the output in the UI of the "cl_showpos 1" command

In image 3.3.2.1 two problems can already be seen: firstly, from the first row, corresponding to the *name* section, we can see that the UI does not accommodate the new elements in any way, as the name of the character overlaps with the already existing UI elements; secondly, the font colour is unmodifiable, meaning that the text may become unreadable on a lighter background.

The first issue was not a problem, as that region of the screen is unused by the standard UI, and only the name slightly overlaps, an aspect of no concern to the bot. However, the second issue would eventually pose significant problems.

Two other aspects worth emphasising are that the size of the digits is approximately the same, regardless of the digit (or about 15 x 25 pixels in a standard 1920 x 1080 resolution) and that the position of the displayed information is static.

This second piece of information represented the foundation of the first idea: capturing the corresponding region of the screen, passing it through an algorithm for digit recognition, and reconstructing the final numbers.

The first problem with this approach is that, although digit recognition can be done easily, number reassembling is not as facile. The third piece of information comes in handy at this point: the format of the output. As can be seen from the *x* component of the position (value *-980.00*), regardless of the value of the fraction part, there are always two digits after the decimal point. Although the value before the decimal point may vary in length (the map ranges to values lower than 10.000 in module), each component contains a decimal point and two digits after. Given this information, with the digits, possible minuses and decimal points correctly identified, the number reconstruction process became straightforward.

The first idea that came to mind was to gather a set of images, label them and use a neural network (as the support for such an idea already existed due to the attempt at enemy recognition) to train a model that would predict, given a 15 x 25 RGB image, whether the image represented a digit, a minus, a dot or none of this. With a trained model at disposal, I would use it with a sliding window of size 15 x 25 over two images of size 245 x 25 (an approximative cut able to capture the whole position and angle regardless of the value of the coordinates) that correspond to the two set of coordinates, get the symbols and reconstruct the numbers. Even though the idea seemed reasonable, it had two issues: there existed no dataset that I could use, so I would have to gather the images and label them, and it was out of proportion; the size and font of the digits remained invariant to any factor. Anti-aliasing could influence the symbols, but only slightly. Training a whole network would be meaningless, as a simple pattern matching against precomputed templates would be enough.

Therefore, I refined the idea; the sliding window described above was in order. However, instead of training a whole network, I enhanced 3 images (image 3.3.2.2a-c) to use only the full-white pixels for the templates. I created label files for these images, containing the given label of the template and its starting point on the X axis (e.g. the 5 in image 3.3.2.2.a starts at pixel 25).

image 3.3.2.2 a-c enhanced images for the template digits

By only using the full-white pixels for the digits, I could simply modify the templates by editing the training images, to correct possible errors.

The templates were represented through 2-dimensional LinearArrays that had the following property: the white pixels corresponded to positive values, the non-white pixels to negative ones and the sum of all the values in the matrix would be 0 (image 3.3.2.3). The *loadCoordDigitsImages* makes use of the jpg loading mechanism used for the loading images for the first enemy-detection idea.

```
auto loadTemplates(std::string const& folderPath) -> void  {
  static constexpr auto imageCount = 3;
  for (auto dataSet : ImageDataSet<...>  = gabe::utils::data::loadCoordDigitsImages<TemplateDigit>(folderPath, imageCount);
       auto const& e : ImageDataPoint<...> const &  : dataSet.data()) {
    Image activatedImage {};
    for (auto lIdx = 0; lIdx < imageHeight; ++lIdx) {
      for (auto cIdx = 0; cIdx < templateWidth; ++cIdx) {
        activatedImage[lIdx][cIdx] = isActive(e.data[0][lIdx][cIdx], e.data[1][lIdx][cIdx], e.data[2][lIdx][cIdx]);
      }
    }
    auto activatedPixels : LinearArray<...>::float  = activatedImage.accumulate(.0f, [](float x, float y) -> float  { return x + y; });
    for (auto lIdx = 0; lIdx < imageHeight; ++lIdx) {
      for (auto cIdx = 0; cIdx < templateWidth; ++cIdx) {
        _templates[e.label][lIdx][cIdx] =
            isActive(e.data[0][lIdx][cIdx], e.data[1][lIdx][cIdx], e.data[2][lIdx][cIdx])
            ? (1.0f / activatedPixels)
            : (-1.0f / (totalPixels - activatedPixels));
      }
    }
  }
  adjustTemplates(11);
  _templates.serialize("templateDigits.gabe");
}
```

image 3.3.2.3 loading templates based on the training images

Each template will be convolved with the current region given by the sliding window (image 3.3.2.4). The convolution between a template and a correct match would produce a highly positive value, as the white pixels have larger values (255 * 3, a perfectly white pixel). Some convolutions may produce false positives (consider a ‘1’ matching and the left margin of a ‘6’). For this reason, I applied two corrections:

1. the result of the convolution shall be compared against a threshold. This way, a partial matching would no longer be a false positive;
2. I ensured that predictions start as close as possible to the first actual digit (the sliding window would, as much as possible, match a digit/minus from the beginning) and then, the sliding window would “jump” several pixels, based on the detected symbol.

These enhancements increased the performance considerably.

Although the prediction results were remarkable, with perfect identification of the coordinates in cases such as images 3.3.2.4a and 3.3.2.4b, the issue presented earlier, regarding the unmodifiable font, can be seen in image 3.3.2.4c.

image 3.3.2.4 a-c two correctly identified examples and a fully misclassified one

Due to the difference in lighting in unfavourable orientations and the background, corresponding images were completely unreadable for the algorithm. Moreover, even slight regions of powerful lighting would heavily damage the algorithm's accuracy.

I attempted to use an error-correction on the prediction based on the last known position; however, the chosen map, de_dust2, proved to have a considerable number of places with powerful lighting, which resulted in abandoning this idea for position reading, as it became completely unreliable.

```
for (auto leftMargin = 0; leftMargin < imageWidth - templateWidth;) {
  auto enhancer : float (float) const  = [](float x) -> float  {
    if (x < 240.0f) return x + 250.0f;
    return x;
  };
  Digit currentDigit = getCurrentDigit(image, leftMargin).transform(enhancer);

  auto currentPrediction : Prediction  = predictionMaker.getPrediction(currentDigit);

  if (lastPrediction.value > currentPrediction.value && lastPrediction.isValid()) {
    predictionMaker.advance(lastPrediction);

    if (lastPrediction.label == 10) {
      signum[resultIdx] = false;
    } else {
      if (predictionMaker.buildNumber(result[resultIdx], lastPrediction.label)) {
        ++resultIdx;
        if (resultIdx > 3) {
          return {{-1024.0f, -1024.0f, -1024.0f}};
        }
      }
    }

    auto marginOffset : int  = predictionMaker.getOffset(lastPrediction);
    leftMargin += marginOffset;
    lastPrediction = {};
  } else {
```

image 3.3.2.5 digit detection through convolutions

The second idea, which was only a hopeful attempt unlikely to succeed, was to use a memory reading application to find the current player's position. I imagined that, despite the numerous mechanisms

implemented by Valve against such cheating, the current player position was a piece of information that did not provide any considerable advantage; therefore, it might be readable without any protection.

The interface for this process was GameConqueror (image 3.3.2.6), a memory scanning program with a UI that allowed searching for values and intervals to identify map-locations where the searched information may reside. Unfortunately, the very first attempt resulted in a failure. Although I managed to locate certain addresses corresponding to the values of the current position in the X axis (I used the values from the "cl_shopos 1" as target), moving the avatar even a bit and searching for the new value provided no matches.

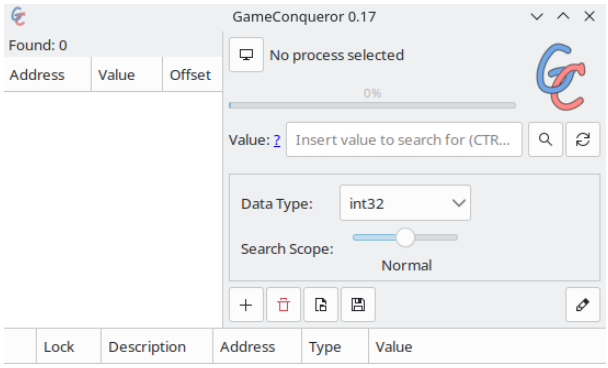


image 3.3.2.6 the GameConqueror interface for memory reading

Although the second idea made no use of the first one, the third, and final solution, did. The *"*cl_showpos 1*"* command proved to be of little help. Yet, a related command, "getpos", was more accommodating. This command did not yield any result to the UI; instead, it printed the current player position and orientation in the developer console. The only issue was that the contents of the developer console were unavailable to my executable and a workaround had to be developed.

Counter-Strike allows, as any executable, passing of command line options at launching [14]. Steam allows us to simplify this process by providing launch options directly in the UI (image 3.3.2.7). Two of these options, which can be seen in image 3.3.2.7, were of interest: "condebug" that "Logs all console output into the console.log text file." and "conclearlog*"* that "Clears the console.log text file on start. Only works if -condebug is set." With these two flags enabled, I could continuously request the current position with the "getpos" command and process the *console.log* file to get thecommand's result. Luckily, the output of "getpos" had a very convenient format: *setpos pos_x pos_y pos_z setang ang_x*

ang_y ang_z; where the *pos_x*, *pos_y* and *pos_z* values correspond to the coordinates and *ang_x*, *ang_y* and *ang_z* correspond to the orientation.

Issuing a "getpos" command by typing it repeatedly would have been infeasible and cumbersome. A better alternative was available: by creating a keybind to the "getpos" command with the developer console and binding it to a single key ('p' in this case), pressing 'p' would result in a "getpos" command being issued.

As Counter-Strike logs a considerable amount of information to the console, the file needs to be constantly parsed in order for the position and orientation to be updated in real-time. Considering this aspect, a separate thread is responsible for constantly parsing the file and updating the necessary information in a multithread-safe memory region (image 3.3.2.8).

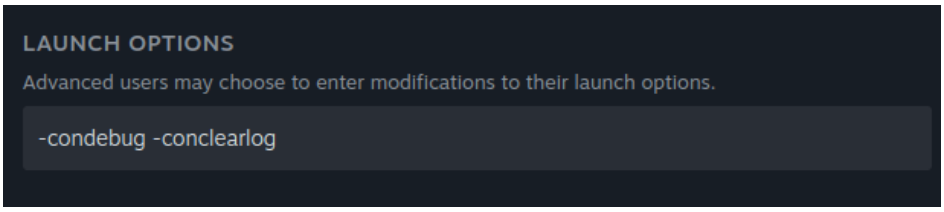


image 3.3.2.7 passing launch options to Counter-Strike

There some aspects to note here:

1) the path to the *console.log* file depends on the location of Counter-Strike on the user machine; this requires the user to pass the path to the Counter-Strike root folder at compilation (see section 4)
2) constantly reading the file proved to block the thread; for unknown reasons the only way for the thread to stay up-to-date with the contents of the file was to close the file upon reading the EOF marker and reopen it, with the mention that the current file position was remembered to prevent rereading the entire file.
3) in line 49 in image 3.3.2.8 a synchronisation with another thread can be observed; this will be discussed in section 3.4
4) the thread directly updates the memory region with the new position and orientation; although this does imply tight-coupling between the PositionReader component and another, yet to be described component. This choice was made for efficiency reasons.

```
auto mainLoop() -> void {
  constexpr auto setPosOffset = std::string("setpos").length();
  static char currentChar;
  static std::string currentString;

  if (feof(_pIn)) {
    auto lastPos = ftell(_pIn);
    fclose(_pIn);
    _pIn = fopen(_filePath.c_str(), "r");
    fseek(_pIn, lastPos, SEEK_SET);
  }
  fread(&currentChar, 1, 1, _pIn);
  if (currentChar == '\n') {
    if (auto pos = currentString.find("setpos"); pos != std::string::npos) {
      auto stringStream = std::stringstream {currentString.substr(pos + setPosOffset)};
      Position position;
      Orientation orientation;
      std::string angleDelimiter;
      stringStream >> position.x >> position.y >> position.z;
      stringStream >> angleDelimiter >> orientation.x >> orientation.y >> orientation.z;
      _gameState.set(GameState::Properties::POSITION, position);
      _gameState.set(GameState::Properties::ORIENTATION, orientation);
      if (synchronizer.synchronizationRequired()) {
        synchronizer.handleSynchronization();
      }
    }
    currentString = std::string {};
  } else {
    currentString.push_back(currentChar);
  }
}
```

image 3.3.2.8 parsing the console.log file and updating the given memory region

With a fully functional position reading mechanism, only one information gathering component needed development: the Game State Integration communication component.

3.3.3 Game State Integration communication

As mentioned in section 3.2, Game State Integration exposes real time information about the game, and this happens by sending HTTP requests to a user-defined server. The server is customised via a *cfg* configuration file.

Configuration files are files that Counter-Strike automatically runs at startup. In order for Game State Integration to be enabled, a configuration file must be placed under the game/csgo/cfg directory and follow the “gamestate_integration_yourservicenamehere.cfg” naming format. In my case I used the filename *gamestate_integration_gabe.cfg*.

The configuration file (image 3.3.3.1) specifies two aspects: the server where the HTTP requests that carry the expected information and the payload to be sent. The data is sent in a JSON format analysed thoroughly in [4].

```
"Gabe"
{
 "uri" "http://127.0.0.1:3000"
 "timeout" "5.0"
 "buffer"  "0.1"
 "throttle" "0.1"
 "heartbeat" "30.0"
 "data"
 {
    "map_round_wins" "1"          // history of round wins
    "map" "1"                     // mode, map, phase, team scores
    "player_id" "1"               // steamid
    "player_match_stats" "1"      // scoreboard info
    "player_state" "1"            // armor, flashed, equip_value, health, etc.
    "player_weapons" "1"          // list of player weapons and weapon state
    "provider" "1"                // info about the game providing info
    "round" "1"                   // round phase and the winning team
    "player_position" "1"         // forward direction, position for currently spectated player
 }
}
```

image 3.3.3.1 the game state integration configuration file

The first feature to be implemented was the server. I created a simple server using C style sockets and threads (images 3.3.3.2a, 3.3.3.2b). Each request would generate a new thread running a predefined

```
Server(GameState& state) noexcept(false) : _state {state} {
  _properties.sin_family = AF_INET;
  _properties.sin_addr.s_addr = htonl(INADDR_ANY);
  _properties.sin_port = htons(PORT);
  initialize();
}
auto getClient() -> void {
  sockaddr_in client_addr {};
  socklen_t client_addr_size = sizeof(client_addr);
  int fd;

  if (-1 == (fd = accept(_socketFd, reinterpret_cast<sockaddr*>(&client_addr), &client_addr_size))) {
    throw exceptions::AcceptException();
  }

  createThread(fd);
}
```

```
auto createThread(int fd) noexcept(false) -> void {
  auto pTp = new ThreadParam {_state, fd};
  _threadList.emplace_back(&Server::threadMain, pTp);
}
inline auto initialize() noexcept(false) -> void {

  if (-1 == (_socketFd = socket(AF_INET, SOCK_STREAM, 0))) {
    throw exceptions::SocketCreationException();
  }

  if (int optionToggle = 1; -1 == setsockopt(_socketFd, SOL_SOCKET, SO_REUSEADDR, &optionToggle, sizeof(int))) {
    throw exceptions::SocketOptionException();
  }

  if (-1 == bind(_socketFd, reinterpret_cast<sockaddr*>(&_properties), sizeof(sockaddr))) {
    throw exceptions::SocketBindException();
  }

  if (-1 == listen(_socketFd, 1)) {
    throw exceptions::ListenException();
  }
}
```

image 3.3.3.2 a-b C-style server to accept HTTP requests

function (image 3.3.3.3). Following this, I needed to parse the incoming text as a HTTP request (implementation in image 3.3.3.4).

The request headers served no purpose; the body was the main object of interest. As stated, the body was in JSON format and for parsing, I used the CDS library [8] to convert the data into a programmatically processable representation.

We reuse the same idea as in the position reading component: direct updates of the corresponding memory with the newly acquired information. As seen in image 3.3.3.4, each thread contains a reference to the target object, an object whose read and write operations are multithread-safe, and updates it based on the JSON information.

```
static auto threadMain(void* pParam) -> void* {
  static std::mutex myMutex {};
  static auto cout = std::ofstream("fisier.out");

  auto threadParam = std::unique_ptr<ThreadParam>(static_cast<ThreadParam*>(pParam));
  HttpMessage response {"Http 1.1/ 200 OK", {}, "ok"};
  Socket const socket {threadParam->fd};

  if (HttpMessage request = socket.read(); request.getBody().starts_with('{')) {
    try {
      auto lg = std::lock_guard {myMutex};
      try {
        auto jsonData = cds::json::parseJson(request.getBody());
        threadParam->state.update(jsonData);
      } catch (cds::Exception const& e) {
        cout << "Error\n";
      }
      cout << request << '\n';
      cout.flush();
      socket.write(response);
    } catch (exceptions::ConnectionTimeoutException& e) {
      log(std::format("Exception {} while receiving http requests", e.what()), OpState::FAILURE);
    }
  }

  close(threadParam->fd);
```

image 3.3.3.3 the thread function executed upon receiving a new request

As the server has to constantly wait for new requests to ensure real-time updates of the game state, a separate thread is responsible for receiving the request and launching other threads in execution to parse the data. This follows the same paradigm as the position reading component and, as we shall see in section 3.4, they belong to the same class hierarchy.

```
auto read() const noexcept(false) -> HttpMessage {
  if (auto bytesRead = ::read(_fd, _pBuf, MSG_SIZE); 0 >= bytesRead) {
    throw exceptions::ConnectionTimeoutException();
  }

  auto pTopLine = strchr(_pBuf, '\r');
  if (!pTopLine) {
    return HttpMessage {"\n", HttpMessage::HeaderType(), _pBuf};
  }
  *pTopLine = '\0';

  auto pHeadersFinish = strstr(pTopLine + 2, "\r\n\r\n");
  *(pHeadersFinish + 1) = '\0';

  HttpMessage::HeaderType headerMap {};
  for (auto pHeaderStart = pTopLine + 2, pHeaderEnd = strchr(pHeaderStart, '\r'); pHeaderStart < pHeadersFinish;
       pHeaderStart = pHeaderEnd + 2, pHeaderEnd = strchr(pHeaderStart, '\r')) {
    *pHeaderEnd = '\0';
    auto pHeaderTitle = strchr(pHeaderStart, ':');
    *pHeaderTitle = '\0';
    headerMap.try_emplace(pHeaderStart, pHeaderTitle + 1);
  }
```

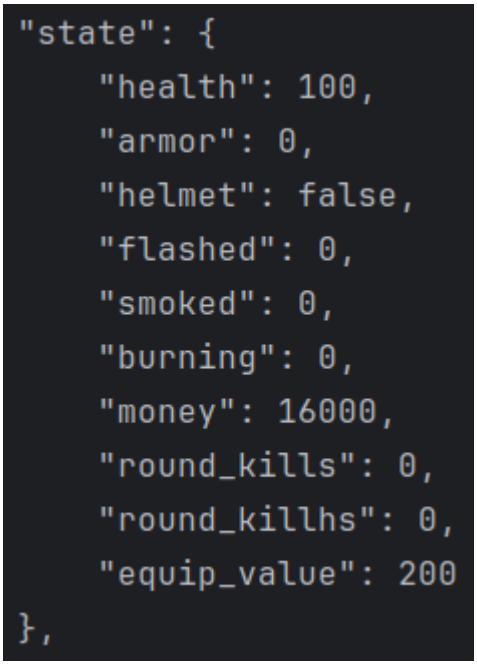
image 3.3.3.4 HTTP format parsing

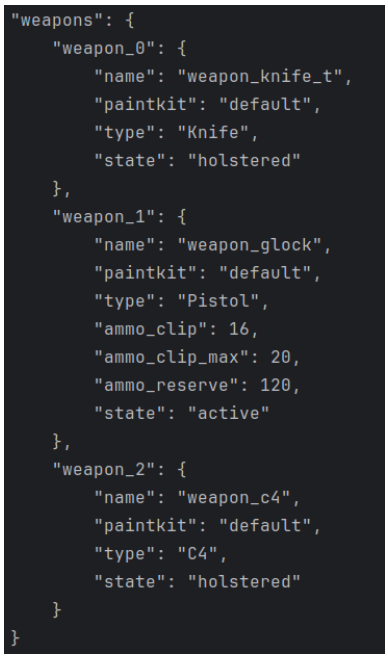
Turning back to the Game State Integration issue. We have a way of extracting the information. However, we still haven't analysed the payload of the HTTP request to see the available information.

As it is used in major tournaments, the payload contains information related to the number of kills, the player's score and other such aspects of no interest to our bot. Of interest, on the other hand, are the following:

1. information about the player state in the *player* element: armour, the presence of a helmet, the money and the health; the first three aspects are of interest during the buying phase. The last one could be used to alter the behaviour of the bot.
2. information about the player's inventory: the payload contains a JSON object, *weapons*, that fully describes the current equipment through multiple elements named *weapon_idx*. Each *weapon_idx* contains information such as the weapon name, weapon ammo, weapon state (whether it is reloading or not / whether it is the active weapon or not). A slight issue with the *weapons* element is that it is not consistent; the absence of a weapon (say rifle) produces a reindexation of the elements. For this reason, the weapon name is crucial for a proper update of the inventory.
3. information about the state of the round in the *round* element: whether the current phase is buying, running or ending. Information about the bomb (if it is planted or defused) also appears in this element after the bomb has been planted; prior to that moment, no information about the bomb is sent.

Examples of all the payloads are in images 3.3.3.5a-c.





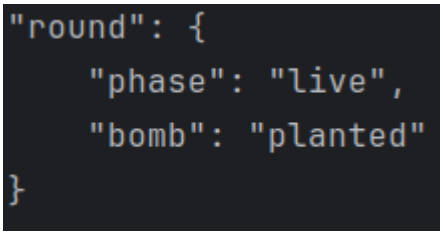


image 3.3.3.5 a-c JSON payload examples

In order to reflect the information received from GSI, I mirrored the three elements into three classes to reflect this separation: the **Inventory**, **Player** and **Round** classes, all of them specialising the base **JsonUpdatable** class (image 3.3.3.6) that makes use of static polymorphism mechanism in C++, as the behaviour is fully known at compile time.

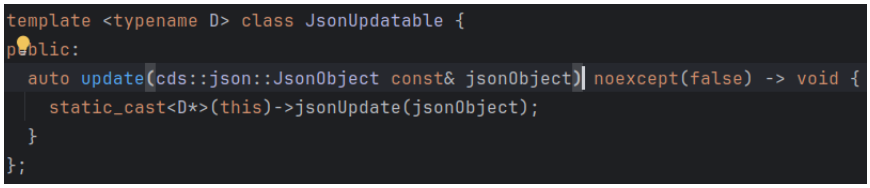
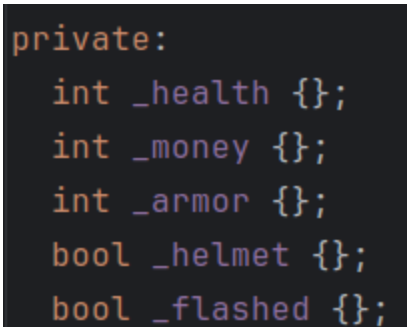
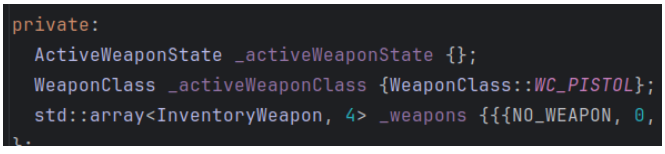


image 3.3.3.6 the **JsonUpdatable** class

The members of the three classes mentioned above can be seen in images 3.3.3.7a-c





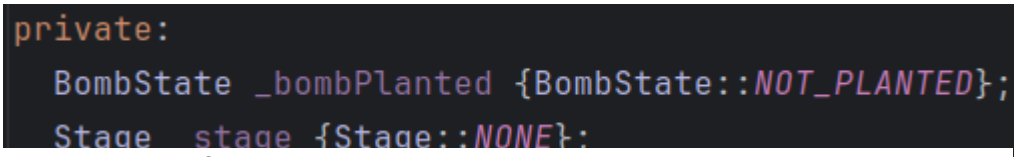


image 3.3.3.7 a-c members of the **Player**, **Inventory** and **Round** classes

As can be seen, the members of the classes mimic the information available from GSI. The only member that needs special attention is *_weapons* from the **Inventory** class. As mentioned previously, the *weapons* element in the payload is an unordered collection of all the weapons held by the player. The inventory has only the knife weapon that is always present; the main weapon, secondary weapon and plantable bomb may be dropped or acquired if the item is missing from the inventory (Counter-Strike also has grenades as throwable items but the bot does not use them, so their presence in the payload is ignored).

Another edge case is the bomb state: as mentioned, the payload does not contain any information regarding the bomb if it is not planted. To maintain a consistent state of the bomb, the existence of the key *bomb* must be checked under the *round* item from the main JSON; the absence of such a key is equivalent to the bomb state being *NOT_PLANTED* (see image 3.3.3.8 for the implementation).

```
try {
  auto const& round = jsonObject.getJson("round");
  _stage = stageMatcher[round.getString("phase")];
  try {
    _bombPlanted = bombStateMatcher[round.getString("bomb")];
  } catch (KeyException const& e) {
    _bombPlanted = BombState::NOT_PLANTED;
  }
} catch (KeyException const& e) {
  log("Exception encountered while updating round state " + e.toString(), OpState::FAILURE);
  throw e;
}
```

image 3.3.3.8 updating the round and bomb state given a JSON; the bomb key may be absent, resulting in the bomb state modifying to *NOT_PLANTED*

With this, the analysis of the Game State Integration component is finished; and this concludes the three components used for analysing the environment and aggregating information.

The last subsection will focus on a different yet essential component: the environment interaction part.

3.3.4 Window Controller

As stated in the beginning of this paper (section 3.0), X11 is required as the display server for the application to function. The reason behind this is that the entire interaction between my application and a running Counter-Strike program is done through the Xlib library which encompasses all the features of X11 in an API.

Communication between X11 and another window is realised through events; raw X11 events were cumbersome, therefore I used a more standardised version: the Xtest header of the Xlib, a header that provides *FakeEvents* that are easier to configure and send over to X11 for processing.

To be able to use Xlib to send an event to a specific window, there is only one requirement: the window ID. Unfortunately, this cannot be obtained with Xlib and another mechanism is required. A non-standard Linux command is needed, *xwininfo*. This command parses and prints the entire window tree generated by X11; after that a simple search through the result for the name of the window ("Counter-Strike 2" in this case) yields the target id. To accomplish this, a simple bash script (image 3.3.4.1a) was used from within my application (image 3.3.4.1b), so the required id is now obtained.

```
while :
do
  result=$(xwininfo -root -tree | grep --ignore-case "Counter-Strike 2")
  if [ -n "$result" ]; then
    IFS=' ' read -r -a array <<< "$result"
    echo $array
    break
  else
    sleep 1
  fi
done
```

```
auto findWindow(std::string const& cmd) noexcept(false) -> bool {
  auto exec = [](char const* cmd) {
    std::array<char, 128> buffer {};
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe {popen(cmd, "r"), &pclose};
    if (!pipe) {
      throw window::PipeOpenException {};
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
      result += std::string {buffer.data()};
    }
    return result;
  };

  try {
    std::string output = exec(cmd.c_str());
    _window = std::strtol(output.c_str(), nullptr, 16);
    log(std::format("Found Counter-Strike 2 window id: {}", _window), OpState::SUCCESS);
    return true;
  } catch (window::PipeOpenException const& e) {
    log("Error: " + std::string {e.what()}, OpState::FAILURE);
  }
  return false;
}
```

image 3.3.4.1 a-b the two components for obtaining the X11 window id of Counter-Strike

Issuing a new event after obtaining the id is trivial: a connection to the X11 server must be opened with the *XOpenDisplay* method and all events must have both the window id and the obtained display as parameters; the other parameters are event dependent. Opening a new connection is not required if one already exists, therefore, the same connection shall be used throughout the program's execution.

To create a stable design, the task of passing events to X11 shall only fall on one class: the **WindowController**. This class shall be responsible for the connection and communication with the X11 server. As the X11 events depend on this communication, every event that is to be sent from my program shall first go through the **WindowController** class. To facilitate the use of the available resources and ensure that no event is missed, the class shall offer a queue for other components of the application to post events that are to be redirected to the X11 server.

Passing raw X events to the **WindowController** is not feasible, mainly due to engineering reasons: X events are represented through function calls of different signatures, therefore lacking a common denominator. I used the *Composite* design pattern to define a hierarchy based on the Event class (image 3.3.4.2) to develop a modular design. A subset of events are basic interactions with the application: mouse clicking, mouse movement, key pressing. Based on those, more complex events can be designed, that in the context of Counter-Strike are seen as a single interaction.

The base **Event** class acts as an interface, providing only one method: *solve(Display* display, Window window)*; This method reflects the relationship to the **WindowController** class, as it requires both a X11 connection, the *display* parameter, and a valid window id, the *window* parameter.

As can be seen from the UML diagram, the primitive events are related to key pressing, mouse moving and mouse clicking. Two more events, **EmptyEvent** and **ScreenshotEvent**, are independent of the rest of the hierarchy. Other events compose one or more of the basic interactions to produce more complex results such as a spray[4] effect or a tap[7]. Some events represent even more complex mechanisms, such as buying new weapons or armour, interactions that need to be fully encapsulated in a single event, and the motivation shall be described in section 3.4.

It is essential to mention about the events that they represent bottlenecks in our application. The reasons behind this are that, in order for our bot to have a human-like behaviour, the speed of its aim must be within reasonable parameters. Although a detection is successful, aiming directly at the enemy without any delay is an inhumane action. Second of all, the speed at which a window is able to process continuous inputs is far less than the computation capabilities of an application. Due to these reasons, delays must be introduced in the process of treating **Events**.

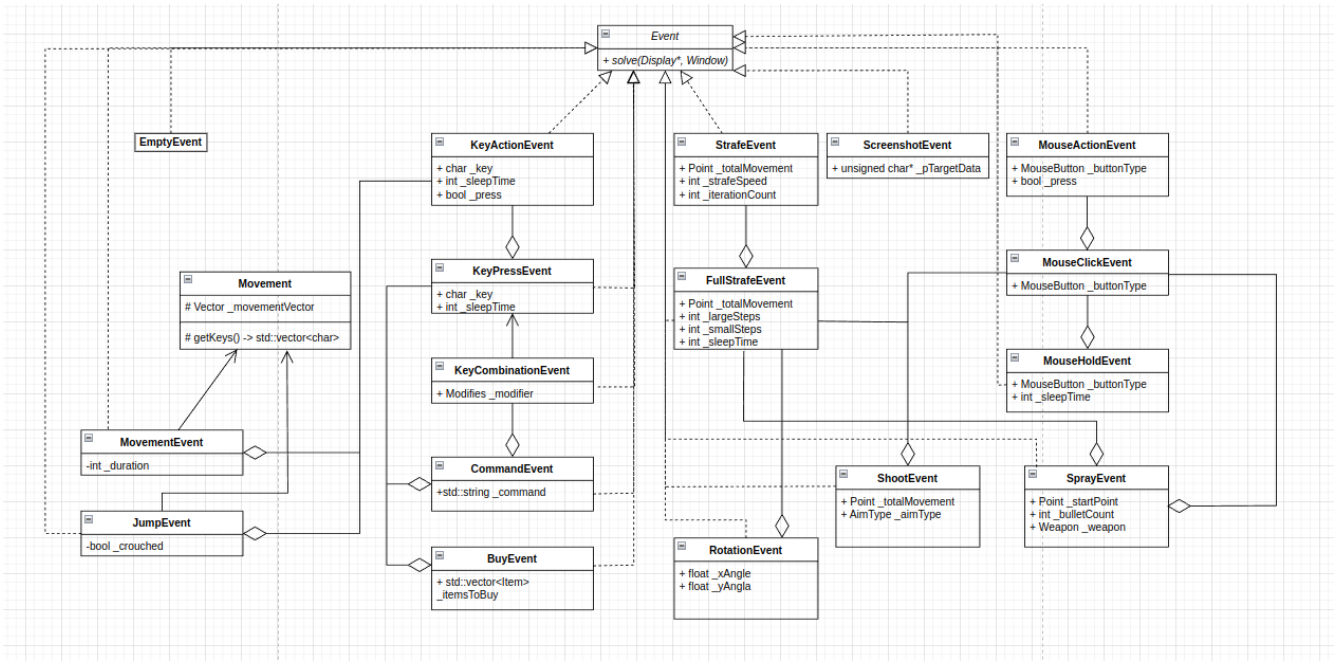


image 3.3.4.2 the hierarchy of **Events**

Having concluded the digression about the **Event** class hierarchy, we return to the **WindowController** class. The class's behaviour is straightforward: expose the aforementioned queue and process events as they arrive by providing a connection and the window id for treating the events. Additional mentions need to be taken into consideration.

Firstly, we have seen that events represent bottlenecks; to prevent computation from being slowed down too heavily, the **WindowController** class shall follow the multithreaded model of the **PositionReader** and Game State Integration communication classes. The **WindowController** class was, actually, in the development process, the first class whose responsibility was placed in a separate thread. The other two classes, described earlier in this paper, were only later brought under this idiom. To ease the implementation of this behaviour, another class was introduced to encapsulate the responsibility of a multithreaded class: the **Runnable** class (image 3.3.4.3).

This class uses the static polymorphism mechanism of C++ and has two functions: to ease the creation of a new thread of execution and to allow graceful termination through the *_stopped* member and *stop* method.

```cpp
template <typename D> class Runnable {
public:
  auto run() {
    auto endlessLoop = [this]() {
      while (!_stopped) {
        static_cast<D*>(this)->mainLoop();
      }
    };

    _thread = std::jthread {endlessLoop};
  }

  auto stop() -> void {
    _stopped = true;
    _thread.join();
  }

private:
  std::jthread _thread {};
  bool volatile _stopped {false};
};
```

image 3.3.4.3 the **Runnable** class

Secondly, synchronisation mechanisms are a necessity in a multithreaded application. As we have seen in section 3.3.2, the **PositionReader** class presented a synchronisation mechanism that we are now going to describe.

The idea behind it is based on a mechanism similar to a barrier: it ensures that given two threads that share a **Synchronizer** (image 3.3.4.4), one can wait for the other to reach a certain point in its execution. By calling the *requestSynchronization* method, the synchronizer thread suspends its execution until the synchronised object reaches a synchronisation point in its execution (image 3.3.2.8): having parsed a

```cpp
class Synchronizer {
public:
  Synchronizer() = default;
  Synchronizer(Synchronizer const&) = delete;
  Synchronizer(Synchronizer&&) noexcept = delete;

  auto requestSynchronization() -> void {
    std::unique_lock lockGuard {mutex};
    synchronize = true;
    conditionVariable.wait(lockGuard, [this] -> bool  { return !synchronize; });
  }

  auto handleSynchronization() -> void {
    std::unique_lock lockGuard {mutex};
    synchronize = false;
    conditionVariable.notify_all();
  }

  [[nodiscard]] auto synchronizationRequired() const -> bool { return synchronize; }

private:
  bool synchronize {false};
  std::mutex mutex {};
  std::condition_variable conditionVariable {};
```

image 3.3.4.4 the **Synchronizer** class

new set of coordinates, the “synchronisee” is ready to signal the sleeping thread that it can resume execution (in section 3.4 we shall see the motivation behind this decision). Synchronisation uses a simple mutex and a condition variable to prevent busy waiting.

The synchronisation mechanism described above is also present in the **WindowController** class (image 3.3.4.5), but this aspect, too, shall be analysed in section 3.4 as the entire context is needed.

```
auto mainLoop() -> void {
  try {
    checkWindowState();
  } catch (window::DisplayOpeningException const& e) {
    log("Error: " + std::string(e.what()), OpState::FAILURE);
  }
  if (_eventQueue.empty() && _synchronizer.synchronizationRequired()) {
    _synchronizer.handleSynchronization();
  }
  if (_windowState.focused && !_eventQueue.empty()) {
    auto event = std::move(_eventQueue.front());
    _eventQueue.pop();
    event.get()->solve(_display, _window);
  }
}
```

image 3.3.4.5 the main loop of **WindowController**

The last aspect worth mentioning about **WindowController** is that it maintains an internal state of the windows focus: if the target window loses focus, events are no longer treated. This introduction was used to prevent keyboard and mouse events from being continuously sent by the X11 server and have a simple mechanism in place for pausing the application.

This concludes the analysis of the main components. Following this, the next section shall describe the orchestration process of these components and the main class responsible for interacting with them.

3.4 The bot engine

This section describes how the previously analysed components are assembled to deliver the end-product, the Counter-Strike bot.

3.4.1 The GameState

The first step in aggregating the components of the bot was to create a container class that encapsulates all the available information. This class was named suggestively **GameState** (image 3.4.1.1).

```
private:
  SharedPosition _position {};
  SharedOrientation _orientation {};
  SharedImage _image {};
  Inventory _inventory {};
  Player _player {};
  Round _round {};

public:
  Map const map {};

  Map::NamedZone targetZone {Zone {}, Map::ZoneName::NO_ZONE};
  std::vector<Zone> currentPath {};
  Position nextPosition {};

  utils::BoundingBox enemy {utils::sentinelBox};
```

image 3.4.1.1 the **GameState** class members

The *_position* and *_orientation* members reflect the contents provided by the **PositionReader** component, the *_inventory*, *_player* and *_round* members are linked to the Game State Integration communication component and the *enemy* member is obtained from the **ObjectRecognition** class.

The other members reflect the following information:

- *_image*: the current image obtained from the **ScreenshotEvent** that shows the current screen; it is a 1920 x 1080 RGB image
- *map*: an internal representation of the de_dust2 map; a deeper analysis is made in the next subsection
- *targetZone, currentPath, nextPosition*: all of these members are computed by the engine and dictate the movement of the bot

As mentioned in the beginning, this class is a simple aggregator of all the information. It is used by the supervisor class (**Engine**) to provide a communication mechanism between the various components.

3.4.2 The Map

As mentioned in section 3.1, the game provides no information about the map. However, by creating an internal representation based on my experience and knowledge of the game, the player position can be used to identify the current map zone. To obtain the necessary representation, I mapped the A-side and middle side of the de_dust2 map by using **Zones** (image 3.4.2.1).

```
struct Zone {
  struct SubZone {
    Volume volume;
    bool accessible {true};
    std::pair<int, int> indices {};

    auto operator<=>(SubZone const& other) const { return volume.operator<=>(other.volume); }
    auto operator!=(SubZone const& other) const -> bool { return volume != other.volume; }
  };

  Volume volume;
  std::vector<Volume> obstacles {};
  std::vector<Volume> hidingSpots {};
```

image 3.4.2.1 the **Zone** class used to map de_dust2 in an internal representation

Each zone is described using a **Volume**, where a **Volume** is specified in 3 dimensions using two opposite corners, making every zone a rectangle when projected to the *XoY* plane. As we shall see, the *Z* axis will not be of much interest to us as the bot does not make use of the elevation in the current implementation. As the information about the obstacles is also unavailable, they were also mapped, still as volumes. The last member, *hidingSpots*, refers to subzones within the main zone that represent points of camping[9] for the bot after planting the bomb. The behaviour and use of the **SubZone** class will be examined later, in the movement segment of the engine.

To facilitate access to the **Zones**, each zone was assigned a name (image 3.4.2.2 shows all the mapped zones)

```
enum class ZoneName {
  NO_ZONE, T_SPAWN, T_SPAWN_EXIT, T_SPAWN_TO_LONG, T_DOORS, DOORS_CORRIDOR, LONG_DOORS,
  OUTSIDE_DOORS_LONG, NEAR_DOORS_LONG, PIT, FAR_LONG, A_SITE_LONG, RAMP, TOP_OF_RAMP,
  GOOSE, A_SITE, TOP_MID, OUTSIDE_TOP_MID, T_SPAWN_TO_MID, BUNELU, MID_TO_SHORT,
  T_TO_SHORT, SHORT_CORRIDOR, SHORT_STAIRS, SHORT_ABOVE_CT, SHORT_TO_A
};
```

image 3.4.2.2 the name of the mapped zones

Another element of the **Map** class are the transitions. **Transitions** (image 3.4.2.3) are the basis for constructing a directed graph with the **Zones** representing nodes and the **Transitions** representing edges.

The graph must be directed, since a transition from zone A to zone B does not imply that from zone B there exists a direct movement back to A. Using this graph, the bot can determine the path required to reach a target zone based on his current zone. The *movement* component of a transition represents whether or not a particular action is necessary to transition from one zone to the other (jumping, crouching) while the *transitionArea* member is used to specify whether transition between two zones must happen through a specific area. However, by default, the transition between two adjacent volumes can occur by passing through the common edge of the two volumes.

```
struct Transition {
  Zone zone;
  RequiredMovement movement;
  Volume transitionArea {};
};
```

image 3.4.2.3 the **Transition** class within the **Map**

As stated in requirement 3.1.3, the bot needs to be able and predict possible camping spots to increase his chance of eliminating the enemy. This aspect was also mapped by hand, as the camping spots remain, most of the time, unchanged.

Having all the elements required to describe the map, I used the “cl_shopos 1” command to identify the borders of each zone, map the obstacles and the zones where enemies may camp and completed transcribing the de_dust2 map into a favourable format (image 3.4.2.4).

```
std::map<Zone, std::vector<Transition>> _transitions;
std::vector<NamedZone> _zones;
std::map<Zone, std::vector<Position>> _watchPoints;
```

image 3.4.2.4 members of the **Map** class that describe the zones, transitions and enemy camp spots by zone

The **Map** class was defined to provide ease of access, such that there is direct access from a **ZoneName** to the corresponding zone and from a **Zone** to its neighbouring regions. As seen in the type of the *_transitions* member, the map **Transitions** do not use the zone name: maintaining information which is used for ease of access twice in the class would be redundant.

3.4.3 The Decision Trees

Decision making during a match involves the player making several decisions every second: where to move next, where to aim, whether to shoot or not, whether to plant the bomb. All of these decisions are made, dependent on one another: moving while shooting at the same time is not a good idea, as the aim becomes flawed during movement.

To replicate this behaviour, the bot shall make use of **DecisionTrees** to make decisions. **DecisionTrees** are data structures that take into consideration the current **GameState**, weigh the available options, and produce a decision. In the context of our application, a decision is represented with an **Event**, as this reflects an in-game action of the bot. Therefore, every **DecisionTree** is continuously evaluated and a corresponding **Event** is generated. In truth, some **DecisionTrees** do not produce an event, as they only update the **GameState** and other intermediate data. However, for the uniformity of the implementation, the **EmptyEvent** is used as the output of such a tree to represent the lack of an effect in the game.

Each **DecisionTree** is represented through decision nodes. An inner node of the tree shall execute its corresponding task and, given the weights associated with each of its children, choose one of them to activate. Terminal nodes of a tree are the ones that produce the final decisions based on their particular behaviour and the current game state. A flowchart of the entire **DecisionTree** structure can be seen in image 3.4.3.1. Two more decision trees exist. However, they will be examined later, as their behaviour is particular.

As can be seen from the diagram, certain trees make decisions solely on the current game state (**MovementTree**). At the same time, others also have a probabilistic aspect, choosing the child to activate based on the weight assigned (**EnemyDetectionTree**).

The order of evaluation of the trees needs careful consideration: the **EnemyDetectionTree** follows directly from the **ImageCapturingTree**. This feature is crucial, as it ensures the lowest delay between capturing the screen and running the enemy detection algorithm.

Every tree also has an associated *postEvaluation* function that is called after the corresponding **Event** has been generated and passed on.

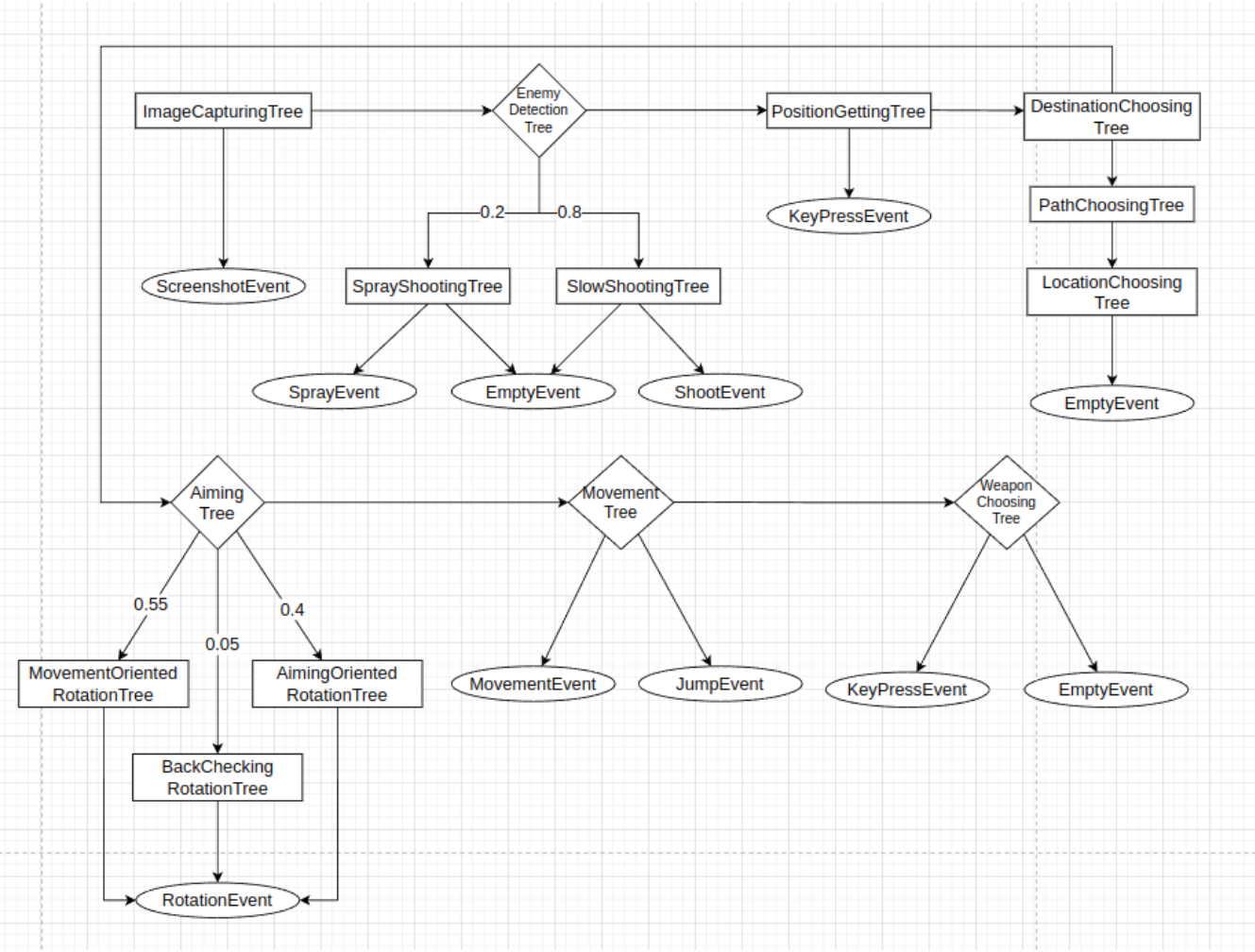


image 3.4.3.1 flowchart of the decision trees

The behaviour of each tree and its children is listed below:

- **ImageCapturingTree**: passes a **ScreenShotEvent** and captures the current screen. An important feature is that the *postEvaluation* function requires synchronisation with the **WindowController**. As will be later described, the entire tree evaluation is done on a single thread, and a synchronisation implies that before continuing with the parsing of other trees, the **ScreenShotEvent** must be produced to ensure an accurate image is currently captured in the **GameState**.
- **EnemyDetectionTree**: calls the **ObjectDetection** component and updates the *_enemy* member of the **GameState** to reflect presence or absence of an enemy on the screen. The two children only get evaluated if an enemy is present. Otherwise, they directly produce an **EmptyEvent** without evaluation.

- **SprayShootingTree**: based on the current weapon and the remaining bullets in the active weapon, it configures and produces a **SprayEvent**.
- **SlowShootingTree**: the distance between the current crosshair and the detected enemy is analysed, and a **ShootEvent** is built and issued.

- **PositionGettingTree**: triggers a "getpos" command by using a **KeyPressEvent** of the created keybind (see section 3.3.2) – in this case a **KeyPressEvent** of the 'p' key. Just as in the **ImageCapturingTree** case, accurate position identification is fundamental. Therefore, the *postEvaluation* method issues a synchronisation request to the **PositionReader** component. It becomes clear here why the synchronisation point in the **PositionReader** component is placed right after updating the position in the **GameState**.
- **DestinationChoosingTree**: chooses the target zone based on the current bomb state: if the bomb is not yet planted, the bot's direction is the *A_SITE* so that the bomb may be planted; otherwise, the bomb is planted, it needs to be defended, therefore the bot shall hide near the side in *GOOSE*; the *targetZone* member in the **GameState** is updated.
  - **PathChoosingTree**: is parametrized on a policy and gives the next location. Based on the Policy, the graph created by the map representation is used to find a path from the current zone (determined based on the current position) to the *targetZone* in the **GameState**. In my implementation, I only used one policy: the **ShortestPathPolicy** (image 3.4.3.2) that represents a BFS traversal.
    - **LocationChoosingTree**: the most complex tree, it is used to navigate the environment. To find a way to traverse the map, the granularity given by the **Zones** is way too large; mapping of smaller regions, however, would be a profoundly cumbersome task. Therefore an automatic mechanism is needed: the **Subzone** class presented earlier. Every **Zone** can be automatically divided into 10 x 10 smaller regions allowing an in-zone algorithm to traverse the current zone towards the destination. Moreover, based on this division, the bot can also detect inaccessible zones as those subdivisions intersecting an obstacle. The algorithm is based on an A-Star navigation of the environment, with really high values associated with the obstacles such that the bot shall never choose a path going through an obstacle. The **LocationChoosingTree** also analyses the game state to determine whether the bot shall target a camping spot or not (see section 3.4.2). In the end the *nextPosition* member in **GameState** is updated to propagate the information.

```
std::map<Zone, Zone> parents {};
std::queue<Zone> zoneQueue {};
zoneQueue.push(startMapZone.zone);
parents[startMapZone.zone] = startMapZone.zone;
while (!zoneQueue.empty()) {
  auto currZone : Zone  = zoneQueue.front();
  zoneQueue.pop();
  for (auto const& zone : const Transition &  : map.transitions().at(currZone)) {
    if (!parents.contains(zone.zone)) {
      if (zone.zone == targetMapZone.zone) {
        path.push_back(map.getZoneByName(targetZone).zone);
        while (currZone != startMapZone.zone) {
          path.push_back(map.getZoneByVolume(currZone).zone);
          currZone = parents[currZone];
        }
        return;
      }
      parents[zone.zone] = currZone;
      zoneQueue.push(zone.zone);
    }
  }
}
```

image 3.4.3.2 a BFS traversal to determine the shortest path to the target

- **AimingTree**: analyses whether an enemy has recently been detected by looking both at the **GameState** to check for a new detection and at an internal state to determine the time of the last detection. If an enemy has been recently spotted, moving the aim would not be advised, as the enemy may still be alive and moving the aim may remove the enemy from the field of view. This tree is also adaptable based on the current state: as observed in the diagram, every child is available based on a probability in the normal evaluation process. In contrast, after planting the bomb the **MovementOriented** and **BackCheckingOriented** children are no longer necessary, as the goal becomes defending the bot and searching for enemies. Given this, planting the bomb disables the two children (image 3.4.3.3).

    - The children of the **AimingTree** all serve a similar purpose: to orient the bot towards a specific goal. This is done using basic 2-dimensional geometry: based on the current position and the target position associated with the goal, it computes the required angle; afterwards, the current orientation of the bot is obtained from the **GameState** the needed rotation angle is computed and it is used to build a **RotationEvent**.
    - **MovementOrientedTree**: the goal (as described above) simply uses the *nextPosition* member as the goal position.

- **BackCheckingOrientedTree**: the rotation angle is directly computed to be 180 degrees (executing a complete turn) to check for the possibility of being flanked[16] by an enemy.
- **AimingOrientedTree**: analyses the current position and, using the **Map**, selects one of the possible watchpoints and sets it as the goal of the rotation.

```
auto postEvaluation() -> void override {
  if (_state.round().bombState() == Round::BombState::PLANTED) {
    for (auto& child : _children) {
      if (dynamic_cast<AimingOrientedRotationTree*>(child.decision.get()) != nullptr) {
        child.weight = 1.0f;
      } else {
        child.weight = 0.0f;
      }
    }
  } else {
    for (auto& child : _children) {
      if (dynamic_cast<AimingOrientedRotationTree*>(child.decision.get()) != nullptr) {
        child.weight = 0.4f;
      } else {
        if (dynamic_cast<MovementOrientedRotationTree*>(child.decision.get()) != nullptr) {
          child.weight = 0.55f;
        } else {
          child.weight = 0.05f;
        }
      }
    }
  }
  DecisionTree::postEvaluation();
}
```

image 3.4.3.3 the **AimingTree** adapting to the current **GameState**

- **MovementTree**: uses the *nextPosition*, the current position and orientation to compute a movement vector that is then used as a customisation parameter for a **MovementEvent** / **JumpEvent**. The distinction between the two events is made by checking whether the current position corresponds to a transition zone registered in the map; based on the movement required to execute the transition, the produced event is customised.
- **WeaponChoosingTree**: the tree analyses the inventory to determine the proper weapon that is to be used. Depending on whether the active weapon and the target weapon match, the tree may produce an **EmptyEvent** as the weapon is already equipped. Another aspect of concern to this tree is related to enemy detection: movement with the knife is faster, but it leaves the bot vulnerable if an enemy is spotted. Therefore, not detecting an enemy allows the **WeaponChoosingTree** to choose the knife to increase the movement.

As stated, two more trees are present in the hierarchy: the **BuyingTree** and the **BombPlantingTree**. They are special because they are **SituationalTrees**, meaning that their evaluation is required only under particular circumstances.

- The **BuyingTree** is only evaluated in the beginning of the round and if it hasn't already been evaluated in the current round to prevent multiple buyings.
- The **BombPlantingTree** is only evaluated if the bomb is present in the player inventory and the current zone is *A_SITE*.

This concludes the analysis of the **DecisionTree** class hierarchy and allows us to examine the final component in the implementation, the main connector between the components, the **Engine**.

3.4.4 The Engine

The entire **Engine** class is an illustration of the *Mediator* pattern. The **Engine** is responsible for creating the threads of the components previously mentioned (**WindowController**, **PositionReader** and GameStateIntegration communicator) and assuring their graceful exit upon termination. The **Engine** also creates the initial **GameState** that is passed to the **DecisionTrees**; for this reason it also builds the entire tree hierarchy so that the same **GameState** is passed to all the trees.

The most important feature of the **Engine** is the passing of events: as mentioned, the WindowController exposes a queue for other classes to use in order to post events. However, only one class is responsible with posting events: the **Engine**. Although the design is multithread, if only one thread enqueues and only one thread dequeues there is no need for a synchronisation mechanism when enqueueing or dequeueing [17]. Although there exists a synchronisation mechanism between the **Engine** and the **WindowController**, its purpose is to ensure that all the events passed so far have been treated.

The sequence in which trees are passed is also dictated in the **Engine** (image 3.4.4.1).

```
auto buildTrees(std::string const& rootFolder) -> void {
  buildImageCapturingTree();
  buildShootingTree(rootFolder + "scripts/objectDetection");
  buildPositionGettingTree();
  buildTargetChoosingTree();
  buildAimingTree();
  buildMovementTree();
  buildWeaponsChoosingTree();
  buildSituationalTrees();
}
```

image 3.4.4.1 the order in which the trees are created gives the order in which they are evaluated

Having examined the **Engine** class as well, this concludes the implementation details of the application.

4 Tests and usage

Application testing implied two parts: first of all, the static elements, meaning classes not related to window interaction, could be tested with a standard framework; and second of all, the dynamic elements, meaning the **DecisionTree** hierarchy, the **Event** hierarchy and the multithreaded aspects were tested by running the application under certain parameters and examining the behaviour.

The static aspects were tested using UnitTest, with Google Test as the preferred framework. Of primary interest in this part of testing was the **LinearArray** class. Considering the clearly defined mathematical results expected from such a class, testing required manual computation of the results before writing the actual tests (images 4.1a and 4.1b).

```
TEST(LinearMatrixTest, Multiplication) {
  auto mtrx1 = larray(larray(1, 2, 3), larray(4, 5, 6));
  auto mtrx2 = larray(larray(2, 2, 2), larray(1, 3, 5));
  auto mtrx3 = mtrx1 * mtrx2;
  auto mtrx4 = larray(larray(2, 4, 6), larray(4, 15, 30));
  ASSERT_EQ(mtrx3, mtrx4);
}
```

```
TEST(LinnearArrayTest, Substraction) {
  auto arr1 = LinearArray<int, 3> {{2, 4, 6}};
  auto arr2 : LinearArray<int, 3>  = larray(1, 1, 1);
  auto arr3 : LinearArray<int, 3>  = arr1 - arr2;
  auto arr4 : LinearArray<int, 3>  = larray(1, 3, 5);
  ASSERT_EQ(arr3, arr4);

  auto arr5 : LinearArray<int, 3>  = arr1 - larray(0, 0, 0);
  ASSERT_EQ(arr5, arr1);

  auto d_arr1 = LinearArray<double, 3> {{1.89, 3.11, 5.23}};
  auto d_arr2 : LinearArray<double, 3>  = larray(1.1, 0.28, 11.53);
  auto d_arr3 : LinearArray<double, 3>  = d_arr1 - d_arr2;
  auto d_arr4 : LinearArray<double, 3>  = larray(0.79, 2.83, -6.3);
  ASSERT_EQ(d_arr3, d_arr4);

  auto eq_arr1 : LinearArray<int, 3>  = larray(5, 5, 5);
  auto eq_arr2 : LinearArray<int, 3>  = larray(2, 3, 4);
  auto eq_arr3 : LinearArray<int, 3>  = larray(-3, -2, -1);
  eq_arr2 -= eq_arr1;
  ASSERT_EQ(eq_arr2, eq_arr3);

  auto scalar : double  = 2.17;
  auto scalar_arr1 : LinearArray<double, 3>  = larray(1.2, 3.33, 2.18);
  auto scalar_arr2 : LinearArray<double, 3>  = larray(-0.97, 1.16, 0.01);
  auto scalar_arr3 : LinearArray<double, 3>  = larray(0.97, -1.16, -0.01);
  auto scalar_arr4 : LinearArray<double, 3>  = scalar_arr1 - scalar;
  ASSERT_EQ(scalar_arr2, scalar_arr4);
  scalar_arr4 = scalar - scalar_arr1;
  ASSERT_EQ(scalar_arr3, scalar_arr4);
}
```

image 4.1 a-b unit tests of LinearArray using GTtest

In addition to testing using GoogleTest, achieving full code coverage was required for maintaining stability during development. Therefore I used the coverage flags of the C++ compiler (-fcoverage-mapping, -fprofile-instr-generate for Clang).

Other classes tested using UnitTests were the neural network related classes: **Layer**, **ConvolutionalLayer**, **NeuralNetwork**. To ensure that the networks implemented using the library were functional, I used two datasets, one for a normal neural network and one for a convolutional neural network, to see if models trained this way could obtain an expected accuracy.

For the standard neural network test, I used a dataset of 210 examples, with 7 attributes and 3 labels. The dataset was obtained from the first semester of the Artificial Intelligence course in the 2023-2024 year, By defining a neural network with two hidden layers, the first of size 5 and using the sigmoid as activation function and the second of size 3 also using the sigmoid, and a final layer with the softmax activation function and the mean squared error cost function I managed to obtain an accuracy of over 90% after 250 epochs across multiple runs. The dataset was divided as 2/3 for training, and 1/3 for validation.

For the convolutional neural network, I used the well known MNIST dataset. The network can be seen in image 4.2. With only 10 epochs and a 0.005 training rate I also obtained an accuracy of over 90%; although the expected accuracy for a CNN on MNIST is about 99%, the lower accuracy in my case is motivated by two aspects: firstly, the low number of epochs, and secondly, the lack of any kind of enhancer for the network.

```
NeuralNetwork<double, ConvolutionalInputLayer<28, 1>, ConvolutionalLayer<32, 3, ReluFunction<>, HeInitialization<>>,
              MaxPoolLayer<2, 2>, InitSizedLayer<100, Layer, HeInitialization<>, ReluFunction<>>,
              InitSizedLayer<10, OutputLayer, UniformInitialization<-1, 1, 1000>, SoftmaxFunction<>,
                             CategoricalCrossEntropyFunction<>>>
    nn;
```

image 4.2 the CNN used for the MNIST dataset

The dynamic aspects were harsher to test, as the behaviour was highly non-deterministic. Debugging issues was also hard, as losing focus on the target window meant that events would no longer get processed. To ease the debugging process, I created a log method, used to display the status of operations (image 4.3). As it can be seen, the log method only has an effect in debug mode; in release mode, the function simply gets ignored.

The developer console and the multitude of available commands were helpful for debugging. By using commands for bot freezing, lengthening rounds or invincibility, I managed to test the bot without the need to go through multiple rounds at a single time, which would have introduced a considerable delay.

To ensure tracking of my progress with the application, I also created a specialised Steam account (image 4.4). Counter-Strike is free-to-play without Account Verification, meaning that the game does not protect matchmaking against cheaters as solidly. However, as I was only playing with bots on a private server,

this aspect had no influence on the testing process. As can be seen in image 4.4, more than 40 in-game hours have been invested in testing the application and gathering the necessary data for examining and mapping the environment.

```
#ifndef NDEBUG
auto log(std::string const& message, OpState opState) -> void {
  auto formatString : const char *() const  = [opState]() -> const char *  {
    switch (opState) {
      using enum gabe::OpState;
      case SUCCESS: return "\033[1;32m";
      case FAILURE: return "\033[1;31m";
      case INFO: return "\033[1;33m";
    }
    assert(false && "OpState undefined");
    return "<undefined operation state>";
  };

  printf("%s %s\n", formatString(), message.c_str());
}
#else
template <typename... Args> auto log(Args&&...) {}
#endif
```

image 4.3 the log method used for debugging

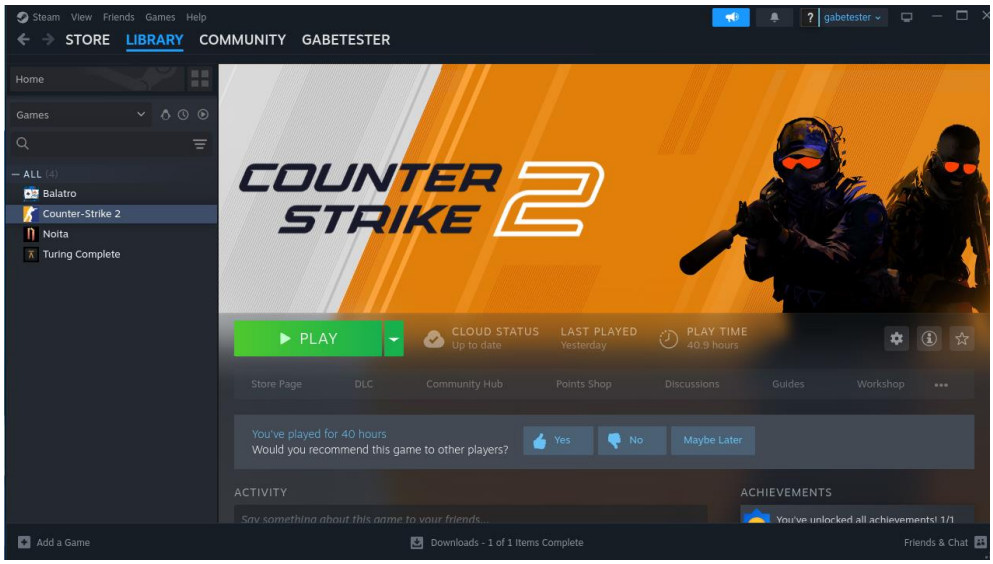


image 4.4 the specialised steam account used for testing

Application usage is relatively straightforward: the main target for Cmake is the *main.cpp* file (image 4.5). As mentioned, it is required to pass the absolute path to the root folder where Counter-Strike is installed, as can be seen in the image. Having compiled the application, it can be directly run. If Counter-

Strike is not opened and the window is not detected (it does not appear in the X11 hierarchy), the application will halt until the window is detected. Afterwards, it will start evaluating the decision trees and send events to the application as previously described.

image 4.5 the main file of the application

5 Conclusions

This project has been a really interesting endeavour for me, as it required knowledge in many subjects, such as Machine Learning, Operating Systems, Multithreaded application development, Software Engineering, Game design and Object Oriented programming. Developing this bot also taught me an important lesson regarding human behaviour: the project initially started with tremendous expectations of having not only a bot with perfect aim, but also with use of considerable mechanics. However, during development, the attempt to formalise what I consider human behaviour proved to be a very difficult task. The way a player makes decisions in Counter-Strike is a highly complex process, that also sometimes relies on one's hunch. It would be impossible for a bot to synthesise and exhibit such a behaviour.

The application also has a few shortcomings that could be worked upon and features that could be added. First of all, let us have a look at the defects:

- the Machine Learning library is functional, however due to a need for fast results in object detection it suffered from several bad decisions during development: currently, extension of the behaviour of the nets is cumbersome; the bright side is that I am fully aware of the modifications required to address the design issues and allow for the library to be further developed
- the object detection model failed; this is perhaps the greatest disappointment of the project. The main problem with this issue is that further knowledge of Machine Learning and neural networks is needed to address this issue. Also, the currently hard to extend design is also a setback, as adding additional features is not currently recommended.
- the application suffers from a deadlock that I have not managed to solve yet due to the complexity of the issue. The problem appears when the thread that issues a KeyPressEvent that in turn generates the "getpos" command and prompts the engine to log the current position in the file halts right before asking for a synchronisation. The second thread manages to parse the file, and as no synchronisation is currently needed, nothing is done. As the first thread resumes, it asks for a synchronisation that will now never arrive, as the last "getpos" that it issued has already been treated.

- error handling is not well done throughout the application and this may lead to unexpected problems.
- movement of the bot does not take advantage of acceleration; given that the bot must, in quick succession, both detect enemies, choose a weapon, a destination and move, the movement process cannot exceed a short duration in time, therefore the bot stops periodically, even if it the pause is in the order of milliseconds. Due to this, the bot builds no velocity from continuous movement, and progresses slower than a normal player through the map.

Other issues of lesser importance in the context of this paper exist. All of them fully documented in the github repository associated with this project at the issues section (https://github.com/StefanPaulet/GABE/issues).

Now, let us analyse the possible enhancements:

- the behaviour of the bot is currently quite rudimentary. As this project wished to be a proof-of-concept, there has yet to be considerable effort towards providing non-rudimentary behaviour to the bot. As can be noticed from the **DecisionTree** flowchart (image 3.4.3.1), there are only a limited number of alternatives to its possible behaviour. The positive aspect is that behaviour augmentation is facile, as it is only required to specialise a tree and add it to the hierarchy. For example, prompting the bot to follow the path that provides the most cover can be easily done by adding a **MovementTree** that has a Policy that encourages a path alongside a wall (this can be easily achieved by customising the heuristic used in the Astar algorithm).
- the bot's adaptability is very limited. As seen in section 3.4, only the **AimingTree** adapts to the current situation and alters the weights of its children. Although it is easy to create an adaptive tree, as it only implies modifying child weights, doing so directly is not the optimal way. A special mechanism would have to be designed in order to allow a more uniform update of the children weights.
- the actions of the bot currently receive no feedback; this enhancement is closely related to the previous one, as providing feedback to the actions is crucial for a good adaptive model. Currently, there is no base for such a feedback providing mechanism; however, it would increase the quality of the bot's performance considerably.

In the end, I would like to thank my good friend Vlad Loghin for all the support provided during the development of this application, especially in the clean and efficient C++ code development section, but

also in helping me bounce ideas off him, offering his input, evaluating the quality of my code, developing the CDS library that was used for JSON parsing library and providing me with library and usage.

6 References

[1] Chenyang Dai - Counter-Strike Self-play AI Agent with Object Detection and Imitation Training

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi – You Only Look Once: Unified, Real-Time Object Detection

[3] https://developer.valvesoftware.com/wiki/Counter-Strike:_Global_Offensive_Game_State_Integration - The official Counter-Strike Game State Integration Wiki

[4] https://www.reddit.com/r/GlobalOffensive/comments/cjhcpy/game_state_integration_a_very_large_and_indepth/ - An in-depth analysis of Game State Integration

[5] https://developer.valvesoftware.com/wiki/Console_Command_List – The list of developer console commands

[6] https://counterstrike.fandom.com/wiki/Counter-Strike_Wiki – The official Counter-Strike Wiki

[7] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software

[8] https://github.com/LoghinVladDev/CDS – Cpp library for JSON parsing

[9] https://eel.is/c++draft/ - the C++ standard draft

[10] https://en.cppreference.com/w/ - C++ documentation maintained by the community

[11] https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf – backpropagation in convolutional neural networks

[12] https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks – convolutional neural networks

[13] https://universe.roboflow.com/hieu-minh-yx9ep/csgo-2-hfa81 – Counter-Strike 2 YOLO dataset

[14] https://developer.valvesoftware.com/wiki/Command_line_options – Counter-Strike 2 launch options

[15] https://github.com/winlibs/libjpeg - libjpeg library repository

[16]https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/ - Xavier and He initialization

[17] Maurice Herlighy, Nir Shavit - The Art of Multiprocessor Programming

7 Glossary

1. crosshair = the indicative icon highlighting the position that the bullets shot from the gun will hit

2. aiming = the process of moving your crosshair to another spot

3. recoil = the rearward thrust generated when a gun is being discharged; this is represented in the game through the weapon being fired becoming inaccurate; the crosshair reflects this as the shots tend to no longer follow the crosshair

4. spraying = continuous fire on a fully automatic weapon

5. spray-pattern = each weapon follows a predefined pattern of recoiling when continuously fired

6. spray-control = the act of compensating for the spray-pattern during the act of spraying

7. tapping = executing single fire precise shots, with enough time between the shots to prevent the effects of recoil

8. headshot = a hit in the head, resulting in increased, most commonly fatal, damage

9. camp(ing) = staying in one position, usually behind cover, and waiting for the enemy to progress

10. wallhack = a cheat allowing the player to see enemies through the walls

11. aimhack = a cheat allowing a players aim to automatically move on a player's head in inhumane speed

12. site = zone where Terrorists can plant the bomb

13. spawn = zone where the teams start at the beginning of the round

14. ace = a single player eliminating the entire enemy team

15. clutch = winning the round while being the only player left alive of your team while the enemies have more members still alive

16. flanking = traversing the map to get behind the enemy and come from an unexpected position

17. VAC = Valve Anti Cheat software

18. 1vs1 = a game mode where each team has only one player