Generating BRIO™ layouts

Alexandre Kings

Supervisor: Tom Spink

Submitted on 27/03/2023

1 Abstract

BRIO™ is a Swedish brand which has been making and selling wooden railway pieces since 1884 [1]. Pieces link together using pin and hole connectors, creating railway tracks for small wooden trains to move on. These toys are particularly popular amongst families with children, who can build up large collections of railway pieces. The aim of this project is to create a tool that makes it easier for people to construct unique BRIO™ layouts using the pieces they have in their collection. The application the author has developed includes a web interface for users to specify their pieces, and allows them to automatically generate closed tracks which they can visualise on a simple 2D display. Procedural generation is used to construct track layouts from sets of pieces.

2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is NN,NNN* words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

3 Contents

4 Introduction

The core objective of this project was to develop a program capable of generating interesting BRIO™ track layouts from user-determined sets of pieces. The requirement for generating complex and interesting layouts is a key aspect of this project, since the aim of this application is to allow users to discover enjoyable and complex layouts which they can build using their set of pieces. To reach this objective, generation of single-loop layouts was first made possible. The possibility to generate multi-level layouts was added, and multi-loop track-generation was ultimately made possible. Throughout this report, different techniques used to improve the program's performance are discussed. Optimisation of the program was a key aspect of this project, aligning with the requirement for generation of interesting layouts – the higher the performance of the program, the better it is at handling larger sets of pieces, and the larger and more complex the possible track layouts generated.

5 Context survey

Procedural generation is commonly used in the field of video games. In early video games, when computational memory was very limited, procedural generation was used as a way of saving large amounts of space. In 1982, *River Raid* used procedural generation to create interesting terrains, which could not have been stored otherwise on a basic 4 kilobyte Atari 2600 cartridge [2]. More recently, other games like the world renowned *Minecraft* [3], and *No Man's Sky* [4], use procedural generation from a single seed number to create immense 3D universes.

The second key usage of procedural generation is to facilitate tedious, repetitive tasks and provide variety. This is the motivation for this program, where procedural generation is used to facilitate the construction of interesting railway layouts. One of the core ideas behind this project is the necessity to generate closed-loop track layouts. The closest kind of procedural generation to this is found in car racing games, where closed loops are generated to mimic real-world racing tracks. Some techniques used for generating such racing tracks include the positioning of sets of control points in a closed loop, with Bezier curves employed to connect these points together [5]. Another technique for generating racing tracks involves the use of a human-like AI agent that races through trial tracks in order to evaluate and improve them [6].

However, the requirement for closed loop tracks is only part of the problem. The set of input pieces specified by the user, and the discrete nature of BRIO™ pieces where each piece has a specific length, curvature, ascent and number of connection points, are important restrictions that need to be taken into account. It is the inclusion of these constraints that make this a one-of-a-kind procedural generation problem. This is, as far as the author knows, the first automatic BRIO™ track generator ever made.

6 Requirement specifications

A set of objectives was initially determined. These objectives are ranked in three categories, *primary objectives* being the most important, followed by *secondary objectives* and then finally *additional objectives*.

Primary objectives

- *An interface should allow users to select track pieces from a list of basic pieces.*
- *A 2D interface should be available to the user to view the tracks generated.*

One of the most important goals of this project is the creation of an interface for users to select pieces and to visualise generated tracks on a 2D display. The option to display generated layouts in 3D was considered at the start of the project. This would have been useful when it came to visualising multi-level tracks, allowing users to clearly see which pieces were placed above others. However, after a discussion with Tom, it was decided that a simple 2D display would be sufficient to show the constructed layouts, and the emphasis was put on the generation of interesting track layouts rather than on displaying the generated tracks in a sophisticated way.

- *An algorithm should generate one or multiple tracks that can be generated from a list of pieces. Whenever possible, the tracks generated should contain at least one closed loop.*

Tracks that contain no closed loops are not considered to be particularly interesting. For this reason, only track layouts containing at least one closed loop were made possible to generate.

- *The tracks should be generated within a reasonable time.*

Efficiency of the track-generating algorithm was a primary concern throughout the development of the program and many decisions were made with an aim to get the program to run faster for large and more complex sets of input pieces.

Secondary objectives

- *The dimensions of the room should be specifiable by the user. The train tracks generated should then fit in the room specified.*
- *Ascending pieces should be added to the list of available pieces in order to allow for the generation of multi-level train tracks.*
- *Track pieces with more than two connectors should be added to the list of available tracks.*
- *Track pieces with a single connector should be added to the list of available tracks.*
- *The connection between any two pieces is not perfect – this allows for the closing of tracks for which the extremities are "close enough" to each other. This should be encoded as part of the track generating algorithm.*

The last secondary objective specifies that the connection between any two pieces is not perfect, and tracks which have ends that are "close enough" to each other should be counted as valid tracks. This makes the constraints on track-generation less rigid, allowing for the generation of significantly more tracks.

Additional objectives

- *More sophisticated pieces could be added to the list of available tracks.*
- *The position of supports to hold tracks not on ground level should be specified to the user.*
- *The track generation time could be decreased by making use of multi-threading if possible.*
- *The user should be able to select the number of loops to be contained in any generated track.*
- *The use of Web Assembly to run the track generating algorithm more efficiently on the frontend of the application could be considered.*

In the current version of the application, the C++ program is compiled to WebAssembly, making for a fully static webpage. This has the advantage of making the application infinitely scalable.

7 Ethics

This project did not involve any ethical concerns. The signed ethics form can be found at the end of this report, in Section 15.

8 Characterization of the BRIO™ pieces

A key aspect of this project was to determine the characteristics of a collection of BRIO™ pieces in such a way that could enable processing by a computer program. The description of pieces is of major importance since this collection of pieces will be used in the rest of the program, both for the generation and display of track layouts. For this reason, this project kicked off finding first a good data structure to represent pieces, and then determining the specific characteristics of an initial set of BRIO™ pieces. This section discusses the way in which a library of BRIO™ pieces was constituted.

8.1 Pieces determined

The piece characterisation tool described in this section was used initially to determine a set of twelve pieces, to which the standard ascending piece and two switch pieces were later added. Currently, fifteen pieces are available for users to choose from. These pieces and their associated characteristics are stored in a single JSON file – called "Pieces.json" – accessible to different parts of the program.

Throughout the report, "ascending piece" refers to a piece which has connectors at different levels, and "switch" is used for a piece with more than two connectors. The only ascending piece made available is the "N" piece, and the "L" and "M" pieces are the two switches implemented.

Table 1 shows the pieces that are currently available, and Figure 1, Figure 2 and Figure 3 show images of some of the implemented pieces.

Table 1: Currently available pieces.

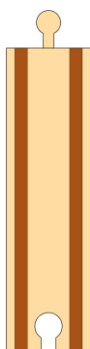| Piece Type	ID
Straight	A, A1, A2, B, B1, B2, C, C1, C2, D
Curved	E, E1
Switch	L, M
Ascending	N

Figure 1: Image of the "medium straight" piece, with id "A" (taken from [7]).
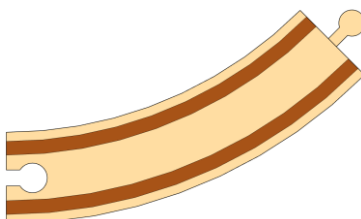

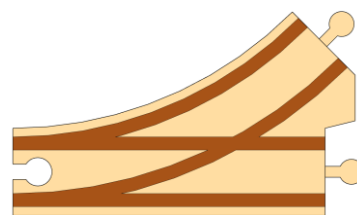Figure 2: Image of the "large curve" piece, with id "E" (taken from [7]).

Figure 3: Image of the "curved switch" piece, with id "L" (taken from [7]).

8.2   Required characteristics

The set of characteristics for each piece had to be carefully determined, to provide a program with enough information to generate layouts using these pieces. More specifically, the track-generating algorithm should be able to determine the following things:

- position and orientation of pieces,
- ability for pieces to connect with each other,
- collisions between pieces.

Also, the data structure representing pieces must be flexible enough to allow for the characterisation of more complex pieces, like switches and ascending pieces. From these requirements, some important attributes appeared.

Each piece is given a list of connectors, each determined by a position, a direction and a type (pin or hole). This allows pieces to connect together, by rotating and translating them so that the two connectors align. The list of connectors can be of any size, allowing for the construction of pieces with one, two, three or more connectors.

8.3   Capturing the shapes of pieces

To allow for the computation of collision between pieces, the general shape of each piece had to be captured. The geometry of pieces had to be modelled in a way simple enough to allow fast and efficient collision calculations. The first option considered was to outline each piece by a single polygon. However, this idea was rejected because the Separating Axis Theorem typically used to determine collisions [8] only works for convex shapes, and curved pieces and switches are not convex but concave. Instead of a single concave polygon, a set of multiple cleverly picked convex ones could have been used for these pieces. However, it seemed complicated to find the right set of polygons to use for some pieces, and having a set of complex polygons for each piece would likely have negatively impacted the performance of collision-detection calculations. It was especially important that collisions could be computed fast, considering that, to place one railway piece, the collision with each of the previously placed pieces must first be calculated.

Another option considered was to approximate pieces' shapes by giving them a set of circles. Collision between two circles is very efficiently and easily determined using Pythagoras' Theorem and comparing the square of the sum of their radii to the square of their distance. However, this idea was rapidly discarded because using circles to approximate rectangular shapes would be unnatural, and a rather large number of circles would have been needed to approximate each shape. Performance would also likely have ended up being negatively impacted from this.

The option chosen was to give each piece a set of rectangles, or 2D Oriented Bounding Boxes (OBBs) [9]. Curves in BRIO™ pieces are 45-degree arcs of different radii [10]. These are relatively small curves and are well approximated using a small set of rectangles.

To help with the determination of OBBs for a set of BRIO™ pieces, it was decided to build a simple graphic tool using the JavaScript Canvas API [11]. This choice of making the tool web-based was implemented so that it could potentially be included later as part of the track-

generating webpage, allowing users to create their own railway pieces and generate tracks with these pieces.

A first version of this tool was made, with the idea that the connectors and OBBs for each piece could be manually positioned, following the shape provided by an image of the piece displayed on the background. An example of the determination of a piece using this tool is shown in Figure 4 (the possibility to determine connectors had not yet been added to the tool).
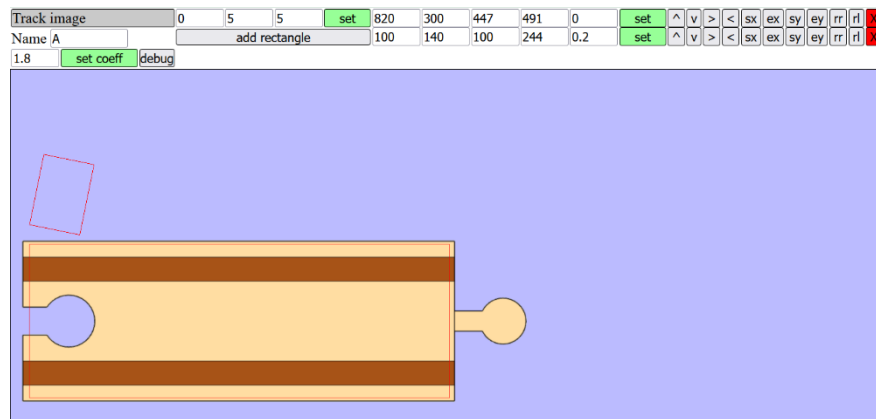

Figure 4: Tool initially developed for the determination of piece characteristics. Example of rectangles (in red) manually positioned, with an image of the BRIO™ A track in the background. Rectangles are determined by 5 numbers – x and y positions of their centre, width, height, and the orientation angle around their centre.

It was however decided that this way of manually determining the OBBs and connectors for each piece would not be satisfactory, for various reasons. Firstly, while the OBBs could be quite easily positioned for simple rectangular pieces, it was difficult to position them properly for curved pieces. Secondly, imperfections in the determination of the orientation of connectors for curved pieces would likely have resulted in strange-looking tracks being generated. The last, and arguably the most important of these reasons, was the scale used for the pieces. The BRIO™ Wooden Railway Guide website [10] provides 2D models of pieces that can be loaded on SketchUp, a free modelling program [7]. Using this, PNG images of the pieces could be obtained. However, this method for sourcing images of the pieces did not conserve scale, and there did not appear to be any trivial way to normalise the scales used for the different pieces. For these reasons, this version of the tool was soon abandoned.

The idea for the next version of this tool was the following: instead of using images of pieces as a basis for determining their characteristics, pieces and their characteristics would be determined in a solely geometrical way, using their known dimensions. Not only does a purely geometrical approach lead to perfectly aligned pieces, it also naturally addresses the scale problem. The dimensions of pieces were found on the BRIO™ Wooden Railway Guide [10].

Each JavaScript Canvas unit is made to correspond to a millimetre of a real piece. Using the fact that all typical BRIO™ pieces are 40mm wide, a function was made to create an OBB of fixed width between two points. Rectangular pieces are then easily determined by two points only, one on each of the piece's connectors. From these points, a single OBB is automatically generated, surrounding the bulk of the piece. The two connectors are oriented along the single axis of the piece. The tool configured in this way also allows to determine the type of each connector, pin or hole. Figure 5 shows an example of the determination of the characteristics of a rectangular piece using the tool described.
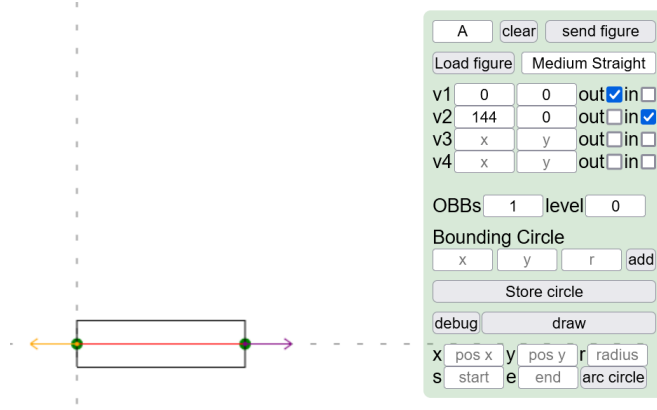
Curved pieces are determined with the same tool, using a set of up to four control points which are used to construct a Bezier curve. This decision of using Bezier curves to determine the characteristics of pieces was made to leave the maximum amount of freedom for constructing complex pieces, if required later in the project. The OBBs for the piece are automatically generated and positioned along the defined Bezier curve. An example of the determination of the characteristics of a curved piece with different numbers of OBBs is shown in Figure 6.
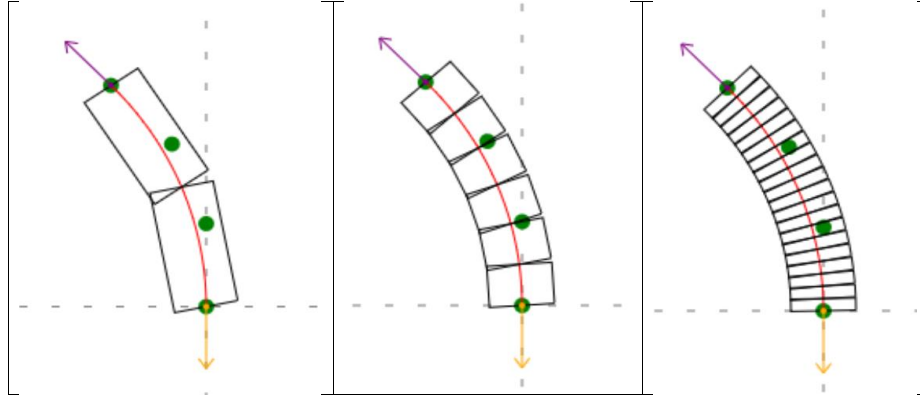

Figure 6: Determination of the "large curve" piece using the Bezier curve plotting tool, with 2, 6 and 20 OBBs. The green points represent the four Bezier control points, and the arrows indicate the position and direction of connectors.

The use of Bezier curves makes it easy to determine simple curved pieces using the arc circle approximation formulae. For an arc of angle β starting at point $P_1 = (x_1, y_1)$ and ending at point $P_4 = (x_4, y_4)$, the coordinates of the two Bezier control points are given by :

$$P_2 = (x_1 + k * r * sin(β/2), y_1 - k * r * cos(β/2))$$

$$P_3 = (x_4 + k * r * sin(β/2), y_4 + k * r * cos(β/2))$$

Using $k = 0.55191496$, an approximation of an arc circle with an error of $1.96 * 10^{-4}$ in the radius is obtained [12], a degree of error which was considered perfectly acceptable in this case. Figure 7 shows an example of a Bezier curve obtained using these formulae.
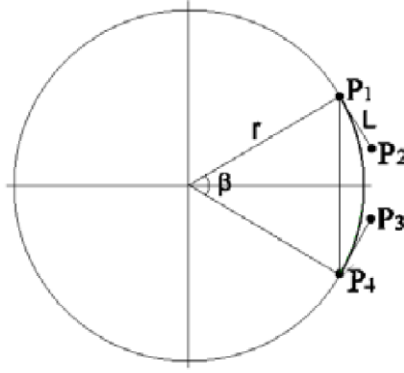
Figure 7: Approximation of an arc circle using a Bezier curve [12]. P1, P2, P3 and P4 are the four Bezier control points used to approximate the arc.

The larger the number of rectangles, the better defined the piece (see Figure 6), but also the longer it will take the program to compute collisions between pieces. It was decided that two OBBs would be sufficient to determine the shape of curved pieces, while ensuring an acceptable degree of precision. This decision was made to keep the program as efficient as possible.

8.4   Piece splitting into parts

To account for switches, it was decided to give each piece a set of separate parts, with each part composed of a certain number of OBBs. Switches are then built using two parts, with one of each part's ends coinciding at the position of the first connector. Figure 8 shows the determination of the characteristics of a simple curved switch.
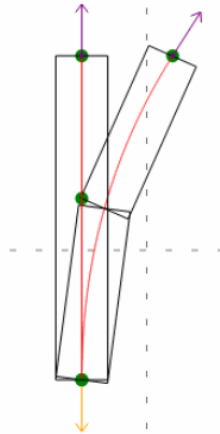

Figure 8: Determination of the characteristics of a switch piece using the Bezier curve plotting tool. Bezier control points are shown in green, and the arrows show the position and direction of connectors. OBBs are represented as black rectangles.

This separation of pieces into a set of parts conveniently allows for the description not only of switches, but also of ascending pieces. BRIO™ pieces can only ascend to multiples of a certain discrete amount. This entailed use of an integer to represent the specific level of each part of a piece. Consequently, a simple straight ascending piece is represented by dividing it into two equal rectangles, one placed at level 0, the other at level 1. Pieces' connectors are given a level in the same way as they are for parts.

8.5 Bounding circles

Later in the program, a bounding circle needed to be added around the bulk of each piece. This addition was made to make collision computation between pieces much more efficient – see Section 10.6 for more details on this. To add these bounding circles, the piece characterisation tool was extended, making it possible to draw a circle around each piece by specifying its centre and its radius. Bounding circles were then manually determined for each piece, ensuring that each circle enclosed all its piece's OBBs. An example of a piece's bounding circle is shown in Figure 9.
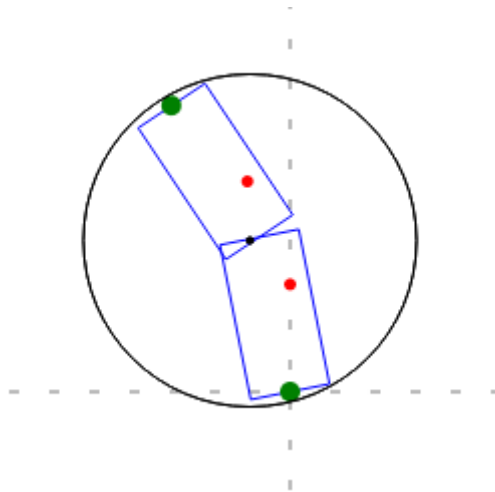

Figure 9: "Large curve" piece, with OBBs shown in blue, Bezier control points in green and red, and bounding circle in black, with its centre a black dot.

8.6 Extension idea

To allow for the construction of special pieces like tunnels or bridges, the tool could be extended to make it possible to add OBBs of arbitrary dimensions. Such pieces are usually composed of a standard piece to which a decoration is added. For example, Figure 10 shows a tunnel piece, which is composed of a straight piece enclosed by a simple cuboid casing. The height of this casing (10.7cm) is enough to prevent the placement of pieces just above it – the difference in height between levels zero and one being 6.4cm [13], [10]. This piece could be simply characterised using two connectors, to which two parts are appended: one with a long rectangle at level zero, slightly larger than the ones used for standard straight pieces, and a second one, positioned at level one, with the same dimensions as the first.

Figure 10: A tunnel piece [9].

16

8.7 Summary of piece characteristics

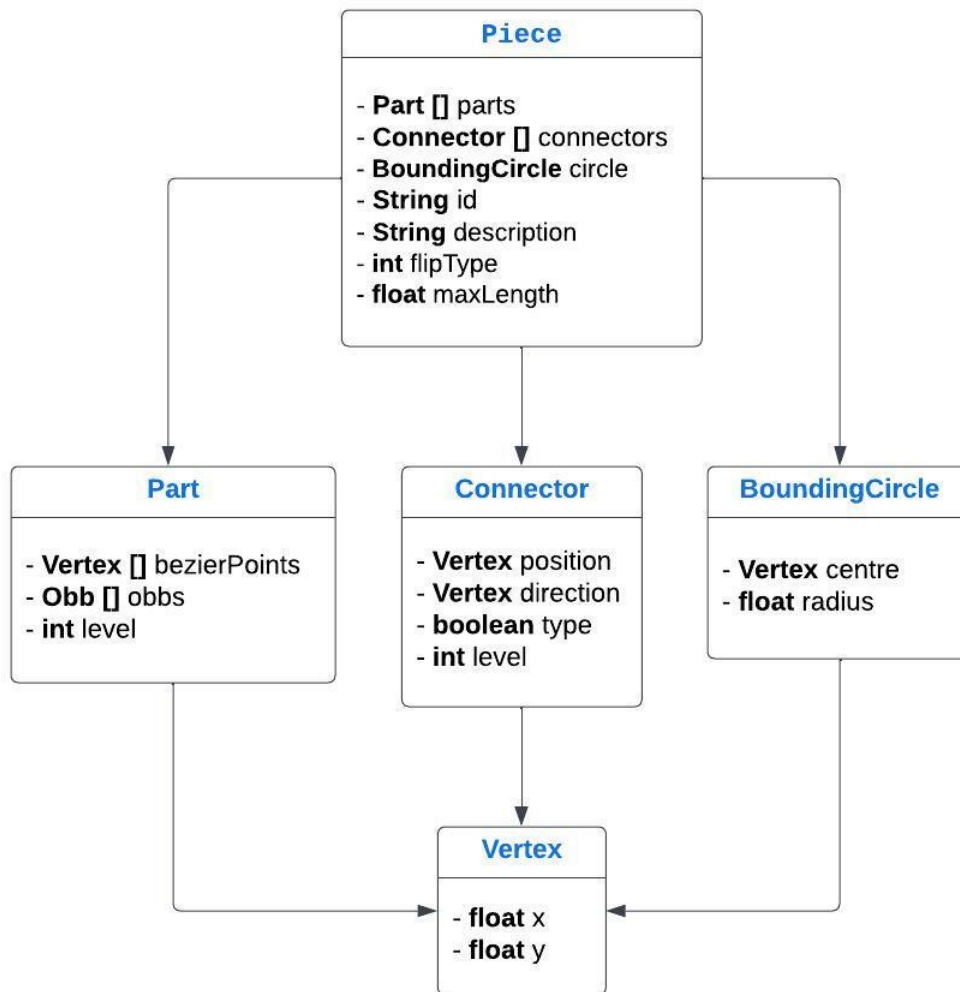Figure 11 shows the different objects used to represent pieces in the "Pieces.json" file.

Figure 11: UML diagram representing the way in which BRIO™ pieces are stored in the Piece.json file.

The Vertex class is simply a pair of two floats representing points and directions in a 2D space. Two piece-attributes present in the diagram have not yet been discussed. The *flipType* integer property tells what kind of algorithm is used to flip the piece around its axis – this is detailed in Section 10.3. The *maxLength* float given to each piece is the Euclidian distance between the two connectors of that piece most distant from each other. This property is essential to the heuristic presented in Section 10.11.

9   User interface

Although the library of pieces was near completion, two important aspects of the project remained: making an algorithm to generate tracks and creating an interface for users to pick sets of pieces and visualise automatically generated tracks. The decision was taken to first implement a simple graphical interface to display BRIO™ pieces and tracks, so that it could later be used to debug the track-generating algorithm. This section details the design decisions made regarding the user interface.

9.1   Tools

An early decision also dictated that this program should be web-based, for users' ease of access. This also allows the program to run on any kind of device, mobile phones and tablets included. The interface is relatively simple, the purpose of it being to allow users to select a certain set of pieces, generate a track from these pieces, and visualise the generated track. Taking this into account, it was determined that no frontend framework would be required, and the choice was made to build the website using standard HTML, CSS and JavaScript.

The page is divided into two simple components: a set of inputs for users to select pieces and specify different generation options, and a display to visualise the generated tracks. Tracks are displayed on a HTML5 Canvas, using the JavaScript Canvas API [11]. This choice was made owing to the familiarity of the author with this API, and its suitability for a 2D application of this nature.

9.2   Representation of generated tracks

Tracks are obtained from the core generating algorithm in JSON format, specifying the position of the pieces' OBBs and connectors. The option to display images of the real BRIO™ pieces at the right locations on the canvas to visualise tracks was considered. For this to work however, a lot of intricate, time-consuming work of texture mapping would have been required for each of the available pieces. A different solution was therefore adopted – the OBBs and connectors themselves were chosen to be displayed, instead of pictures of the real pieces. The initial display is shown in Figure 12.
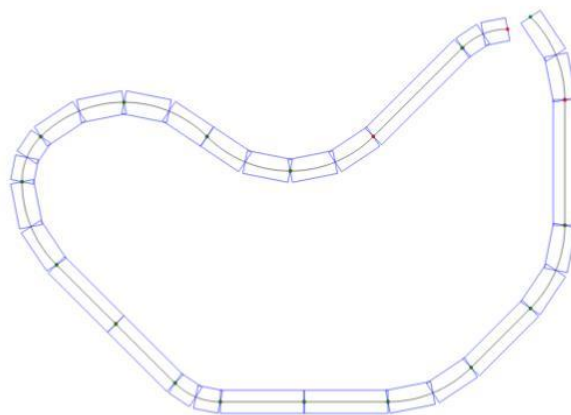

Figure 12: Initial display of a generated track on a HTML5 Canvas. The Bezier curve of each piece was made visible at this stage, but later removed.

Pieces were later colour-coded depending on their ID, and a simple shape was drawn at the location of each pin connector. These improvements facilitate visualisation of the placement of pieces and make it easier for users to reproduce tracks with their real BRIO™ pieces. Figure 13 shows an example of track displayed on screen after these improvements were made.
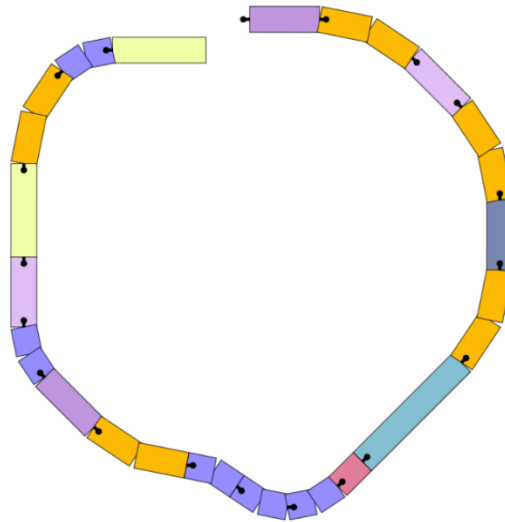

Figure 13: Example of a track displayed on screen, with the addition of connectors and a colour coding for pieces.

Once multi-level tracks were implemented, another colour code was added to the display, allowing users to visualise the level at which pieces were placed. Pieces are still coloured depending on their ID, but outlined in a colour specific to their level. An example of a multi-level track with colour-coded outlines is shown in Figure 14.
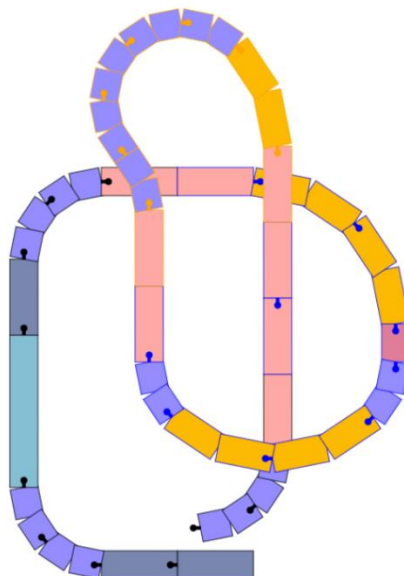

Figure 14: A three-level track, with pieces placed on the ground outlined in black, level 1 in blue and level 2 in yellow.

9.3 Preventing piece overlaps

The JavaScript Canvas API works so that any element added at a position of the canvas hides the elements previously drawn at that position. From the core generating algorithm, the JavaScript frontend code obtains tracks as a list of piece objects, sorted in no particular order. However, the addition of ascending pieces can lead to an issue in the display of tracks. The fact that some pieces were allowed to stand above others, and due to the random order in which the pieces were obtained, some level zero pieces were sometimes painted after higher level ones. This caused some strange piece overlaps in the track and made it difficult to understand which pieces truly lay above others (see Figure 15).

This problem was addressed by making use of a simple technique called Painter's Algorithm [14] – see Figure 16. Pieces are first separated into their component OBBs and connectors, and a single list is then made to contain these elements. The list is then sorted in an ascending order of levels. Finally, each component is displayed on screen, in the order given by the list, enabling the pieces to appear at the right level on canvas. To avoid pieces' connectors from being displayed below their respective OBBs, rendering them invisible, the level of each connector is raised by 0.5 before the list is sorted, which leads to the correct drawing order for the entire track.
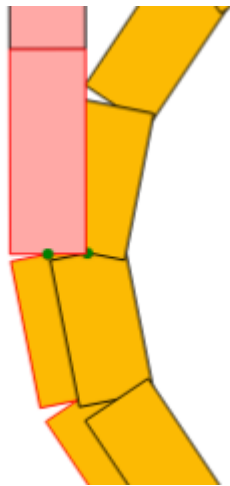

Figure 15: Example of bad display of a portion of multi-level track. Pieces outlined in black are at level zero, and a red outline is used for level one. The two pieces at level one, at the bottom-left of the image, should be painted above the ones at level zero.
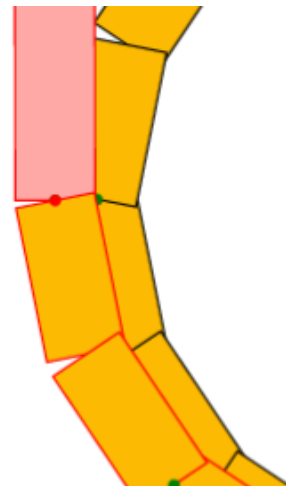
Figure 16: Same portion of track as the one displayed on Figure 11, with the addition of Painter's Algorithm, solving the issue of overlapping pieces.

9.4 Addition of zoom and drag on canvas

The option for users to move on the canvas by means of a clicking and dragging action of the mouse, and to zoom by scrolling the mouse wheel in order to view tracks from closer or further away, was added relatively early in the project. This way of interacting with the canvas to visualise tracks was chosen for its simple and intuitive use. These actions were implemented using a lightweight JavaScript library [15].

It turns out that this implementation of drag and zoom on canvas works well when using a physical mouse. However, touchpads often prove too sensitive and make it hard to position the generated track at the right level of zoom. Also, the dragging and zooming effects do not appear

to function on mobile phones. More research time would be necessary to implement these features in a better way.

9.5 Piece picker

A tool to select sets of pieces was created as a vertical scrollable list, displaying the ID, description and an image of each piece, alongside a basic HTML input in order to select the desired number of corresponding pieces. Images of the pieces were obtained from [10]. This list is shown in Figure 17.

ID	Description	Image	Selection
E	Large curve		6
A	Medium straight		13
B	Medium straight		
C	Medium straight		

Figure 17: Table for selecting pieces from which to generate a track.

9.6 Options box
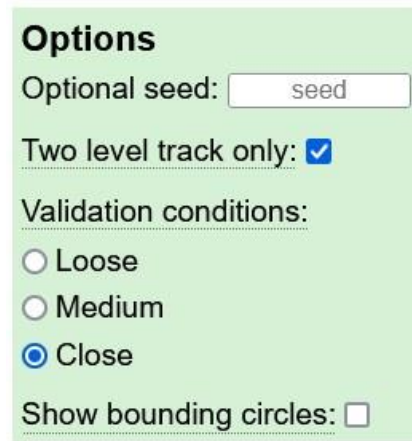
A section of the user interface is allocated for the specification of special generation options. These options are as follows:

- A text input allows users to select a seed,
- A checkbox can be ticked to prevent tracks from exceeding one level above the ground. High tracks can be tricky to build with real BRIO™ pieces, so users might want to use this option to prevent the generation of such tracks.
- Radio buttons allow users to select the kind of validation conditions to use – see Section 10.4 for the definition of validation conditions. Three options can be picked from, either

"loose", for generating highly disconnected tracks, "medium", or "close" for tracks with more connected ends.

- A checkbox can be selected to display pieces' bounding circles.

Figure 18 shows the options box.

Figure 18: Screenshot of the "options box" of the user interface.

9.7 Small improvements

Small improvements on the user interface were made towards the end of the project. These were added following suggestions by my supervisor Tom, who tested the program by building a generated track with real BRIO™ pieces. Tom's testing of the program is detailed in Section 11.3.

One issue found by Tom was that when generating a track from a random seed, there was no way of knowing which seed was actually used by the program to generate the track. This could be frustrating for a user who obtained a nice track from a random seed and would like to recreate it later. For this reason, a message was added to display the seed used for any generated track.

Another issue was that a track generated was lost on refreshing the webpage. Users had to manually re-specify their selection of pieces every time the page was refreshed – this could be annoying for users trying to build a real BRIO™ track following the one displayed on screen. To resolve this, pieces used to generate the last track are kept in local storage, allowing a user very easily to recreate the previous track after a page reload.

The current state of the user interface is shown in Figure 19.

Figure 19: Current state of the user interface, with an example of track displayed.

10 Track Generation

As discussed in the introduction, the core track-generating algorithm is implemented in a C++ program that is separated from the user-interface code. The algorithm works by randomly picking pieces and connecting them to each other until a closed loop is found. This section details the method used for generating BRIO™ track layouts.

10.1 Tools

Java and C++ were initially considered for the implementation of the track-generating algorithm. Both are object-oriented languages, and offer options for multithreading, which had been identified as an important feature at the start of the project. However, potential hindrance on performance caused by Java's garbage collector, and more importantly, C++'s potential for getting compiled to WebAssemlby made the scales tip in favour of C++.
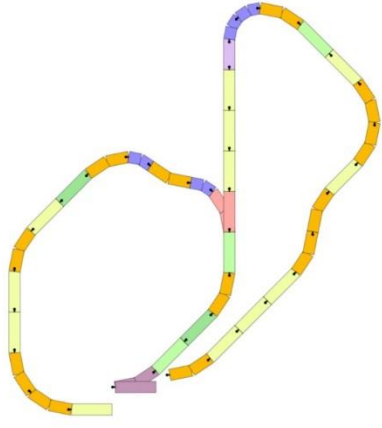
It had initially been decided to run the track-generating algorithm on a backend server, and to setup communication between frontend and backend using a NodeJS API. This worked fine for a while. However, this setup was later changed in favour of a fully static website, with the track-generating C++ program compiled to WebAssembly. Removing the use of a server carries the advantage of making the application infinitely scalable, as it runs directly on users' machines. WebAssembly is supported on all modern browsers [16], so most users should have no problems running the program on their machine.

Emscripten was the compiler used to compile the C++ program to WebAssembly. Certain compilation options are specified [17]. "ALLOW_MEMORY_GROWTH" is used to allow the total amount of memory to change depending on the needs of the program – this is crucial for the program to be able to generate tracks from large sets of pieces. "O2" is specified to enable compiler optimisations. The "preload-file" option is used to provide the program access to the "Pieces.json" file.

Meson [18] is used as a build system to compile the C++ project, and makes it easier to include "JsonCpp" [19] as a dependency, to parse the "Pieces.json" file described in Section 8.1.

10.2 Algorithm description

The generation of track layouts is implemented in a recursive method, inspired by the depth-first search algorithm. At the start, one piece is placed at the centre of an infinite 2D area. The algorithm then selects a second piece and connects it to the first one. At each recursion, a piece gets placed at the end of the track, until either the track is fully generated, or no more pieces can be placed. When the latter happens, the function recurses back to the previous step, picking a different piece to place.

Each individual BRIO™ piece selected by the user is represented as a C++ object. The classes used in the C++ backend are like the ones used in the "Pieces.json" file, detailed in the UML diagram shown in Figure 11. The set of all pieces is kept in a single C++ vector, with a flag on each piece indicating whether the piece is in use or not. It was considered to use two different vectors, one for available pieces, and one for used pieces. This would have led to pieces frequently getting moved from one vector to the other, and the use of a flag seemed much simpler.

For a single-loop track, the generating algorithm works as follows:

1) The first piece is positioned in the centre of the area. Its first connector is kept in memory as the *Open Connector*, and its second connector is kept as the *Validation Connector*. The Open Connector is the connector to which the next piece will link itself, and the Validation Connector is the one that needs to be reached for the track to be considered as closed.

2) The available pieces are shuffled in a random order.

3) The first available piece (previously unchecked) is taken into consideration. If it has a connector that is of the opposite type to the Open Connector, the pieces get linked. Otherwise, the piece is rejected, and back to step 3. It is then checked to see whether the newly placed piece collides with any other placed piece. If the piece collides, it is rejected, and back to step 3.

4) If none of the pieces succeed in step 3, back to step 2.

5) The newly placed piece is marked as used, and its available connector now takes the place of the previous Open Connector. Then, back to step 3.

6) Every time a new piece is placed, the *validation conditions* are tested for that piece. Validation conditions are discussed in Section 10.4.

7) When all the validation conditions are met, the track generation stops, and the completed track is presented to the user.

This algorithm is implemented in a recursive way. This is a natural choice: every time a new piece is placed, the track generation function can be called recursively with a new set of available and placed pieces, and with a new Open Connector. Also, due to the fact that a maximum of one recursive call is made per available piece, a stack overflow should not occur from this recursive function since a maximum of only N stack frames are ever required for a set of N initial pieces.

10.3 Piece positioning

An important part of the track-generating algorithm consists in connecting two pieces together. The requirements for connecting two pieces together are as follows: one of the pieces must already be placed and have an open connector, that is, a connector that is not currently used in a connection with another piece. The other piece must not be placed, and it must have a connector of the opposite kind to the open connector.

The objects containing information on a piece's position in space are its OBBs, connectors and bounding circle. Each OBB contains a set of four vertices, one per corner. Each connector contains one vertex for its position, and one 2D vector for the direction in which it is pointing. A 2D vector class – named Vec2D – is used for representing positions and directions.

To connect the two pieces, the second piece is rotated around its connector until the two pieces' connectors align and are in opposite directions. The piece then gets translated to a position where the two connectors enter in contact with each other.

Vector translations and rotations are necessary to change the position of pieces. Translations are easily done by subtracting vectors together. To rotate a piece, each one of its OBBs and connectors' vertices are rotated using the Vec2D rotation function. This translates the vector's coordinates by the opposite of the rotation point's coordinates. The vector is then multiplied by the following rotation matrix:

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Finally, the vector gets translated back by the rotation point's coordinates, completing the rotation around that point.

Another important aspect of piece positioning is the ability to flip pieces. Most BRIO™ pieces can be positioned facing up or down. However, straight pieces are naturally symmetrical when it comes to flipping. Each piece is thus given a *flipType* attribute corresponding to the way in which it must be flipped. A *flipType* of 0 means that the piece does not need to be flipped – this is the case with straight pieces. A *flipType* equal to 1 is used for simple curved pieces that can be flipped by a simple inversion of their two connectors. This also works for the straight ascending piece. A *flipType* of 2 is given to more complex pieces like switches, which require each one of their vertices to be individually reflected around an axis in order to be flipped.

The ability to flip a piece is implemented in a method of the Piece class. This is shown in pseudo-code:

```
- if (flipType == 0): return; // No flip required.
- if (flipType == 1) {  // Exchanging the two connectors is sufficient.
    - exchangeConnectors(connectors[0], connectors[1]);
- }
- if (flipType == 2) {  // A full reflection is needed
    - for (Connector con : connectors) {
        - con.position.reflect();
        - con.direction.reflect();
    - }
    - for (Part part : parts) {
        - for (Obb rect : part.obbs) {
            - for (Vec2D v : rect.vertices) : v.reflect();
        - }
    - }
- }
```

Where **reflect** is a function of the **Vec2D** class – a class representing a simple 2D vector – that reflects the coordinates of the vertex symmetrically around the x axis. This is done simply by leaving the y-coordinate unchanged and changing the sign of the x-coordinate. Note that any axis of reflection could be used, as long as it is consistent with all the piece's vertices.

This use of a *flipType* property for pieces is very impacting from a performance point of view. Thanks to it, straight pieces are known by the program not to require any flipping, and this avoids the need for certain track-generation branches from being searched twice. However, the

performance gained by making simple curves flip by exchanging their connectors, rather than reflecting each of their vertices around an axis, are in effect relatively small.

10.4 Vario System and validation conditions

BRIO™ pieces are intentionally designed to fit imperfectly with each other – two connected pieces can wiggle slightly. This amount of leeway when it comes to connecting pieces is called the Vario System [10], and it allows loops to close much more easily.

The Vario System is encoded as part of the track-generation algorithm. Each time a new piece is placed, certain conditions called *validation conditions* are checked between it and the first piece's open connector. The latter, due to the fact that it is often checked against the validation conditions, is called the *validation connector*.

Validation conditions include the following tests:

- the Euclidian distance between the two connectors needs to be small enough,
- the two connectors need to have their directions align with each other, within a certain margin of error,
- a minimum proportion of the initial number of available pieces needs to be placed.

As explained in Section 9.6, users are offered the possibility to choose between three different sets of validation conditions to generate a track. These are shown in Table 2. The minimum proportion of placed pieces is defined as 60% of the initial number of pieces, for all the sets of validation conditions.

Table 2: Summary of the different sets of validation conditions.

Validation condition name	Minimum distance (mm)	Angle (rad)
Loose	300	$0.4 * \pi$
Medium	200	$0.3 * \pi$
Close	100	$0.2 * \pi$

Using "close" validation conditions is preferable in most cases, as they lead to tracks that should work for real BRIO™ pieces. However, "medium" or "loose" validation conditions make it easier for the program to construct loops, and can be used in cases where the program does not find any closed loops for a certain set of pieces using the "close" conditions. However, tracks built with "medium" or "loose" conditions may need to be slightly modified by the user for the loops to close.

Figure 20, Figure 21 and Figure 22 show three tracks generated with the same pieces and the same seed, but using different validation conditions. It is noteworthy on these figures that the stricter the validation conditions, the closer the ends of the track are.
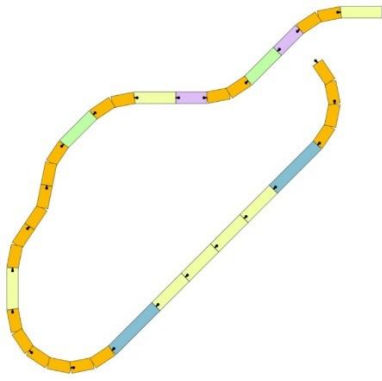
Figure 20: Track built with the set {12E, 7A, 2B, 2C1} using loose validation conditions, seed 4.
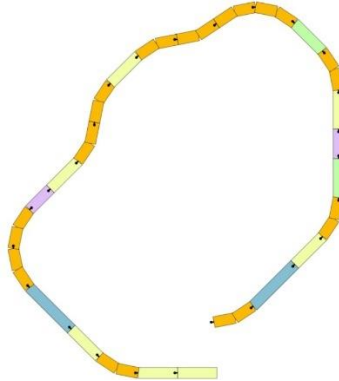
Generation time: 64ms.

Figure 21: Track built with the set {12E, 7A, 2B, 2C1} using medium validation condition, seed 4.
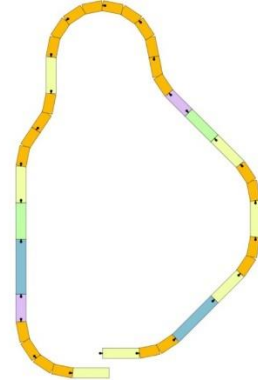
Generation time: 134ms.

Figure 22: Track built with the set {12E, 7A, 2B, 2C1} using close validation conditions, seed 4.

Generation time: 208ms

10.5 Random numbers and seed

Obtaining random numbers is a key part of the track generation. Before generating a layout, the set of pieces is shuffled in a random order. Also, whenever a curved or a switch piece is picked, it is randomly oriented to initially face up or down. This step is key to the algorithm's performance – if random orientation of curves is omitted, all the curves can end up getting placed to face in the same direction, which can result in tracks that spiral around themselves and can't be closed. This inclusion of randomness not only allows for generation of different tracks, but also enormously increases the performance of track-generation – this is the basis of the method for improving performance discussed in part 10.8.

All the random numbers used in the program are obtained from a single random number generator, *std::default_random_engine* [20]. The same random numbers can always be obtained given the same seed, and this is advantageous especially for debugging, as the same track can be re-obtained multiple times. The ability to specify a seed was kept in the final version of the application to make it possible for users to re-generate any particular track which they had previously obtained.

10.6 Collisions

During track-generation, after connecting a piece to the end of the track, the algorithm checks for collisions between the newly placed piece and all the previously positioned pieces. This is essential in preventing pieces from overlapping.

Due to the way in which pieces are represented using OBBs, slight overlaps between connected curved pieces are inevitable – see Figure 23. For this reason, a piece's direct neighbour is ignored when checking for collisions. This assumption works well for the fifteen pieces made available in this version of the program – two pieces taken from this set can never directly collide with each other. Although it is easy to imagine two theoretical pieces with unusual shapes that would collide when connected to each other, after inspection on the BRIO™ website, it seems like any two pieces can always directly connect without colliding.

28
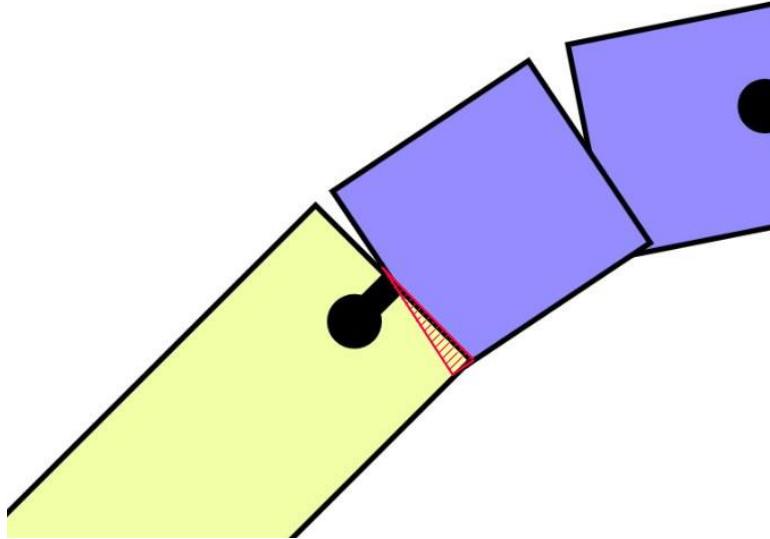
Figure 23: Connected pieces slightly overlapping. The overlap is shown as a red-striped triangle.

To check for collisions between two pieces, each of the pieces' OBBs are checked against each other. The collision between two OBBs is calculated using the Separating Axis Theorem [8]. This theorem states that if a line can be drawn between two polygons without intersecting with either one, then the two polygons do not collide.

It is then possible to compute the collision between any two convex polygons in the following way. Start by taking the projection of all the corners of each of the polygons, along one of the first polygon's sides. Then, take the maximum and minimum points of each polygon along that axis, and check if their maximums intersect with each other – if they don't, the two polygons do not collide. And if the projections of their corners do overlap for all of the polygons' sides, they collide. Figure 24 shows an example of collision computation for two rectangles using this method.

Figure 24: Collision verification for two rectangles. The maximum and minimum of the projections of the rectangles' corners overlap along axis A2, but do not overlap along axis A1. This means that the rectangles do not collide.

Collisions are computed at each piece placement attempt, and against all of the pieces currently in use. This leads to a lot of collision computations, meaning that the efficiency of the collision verification function is critical to the overall efficiency of the program.
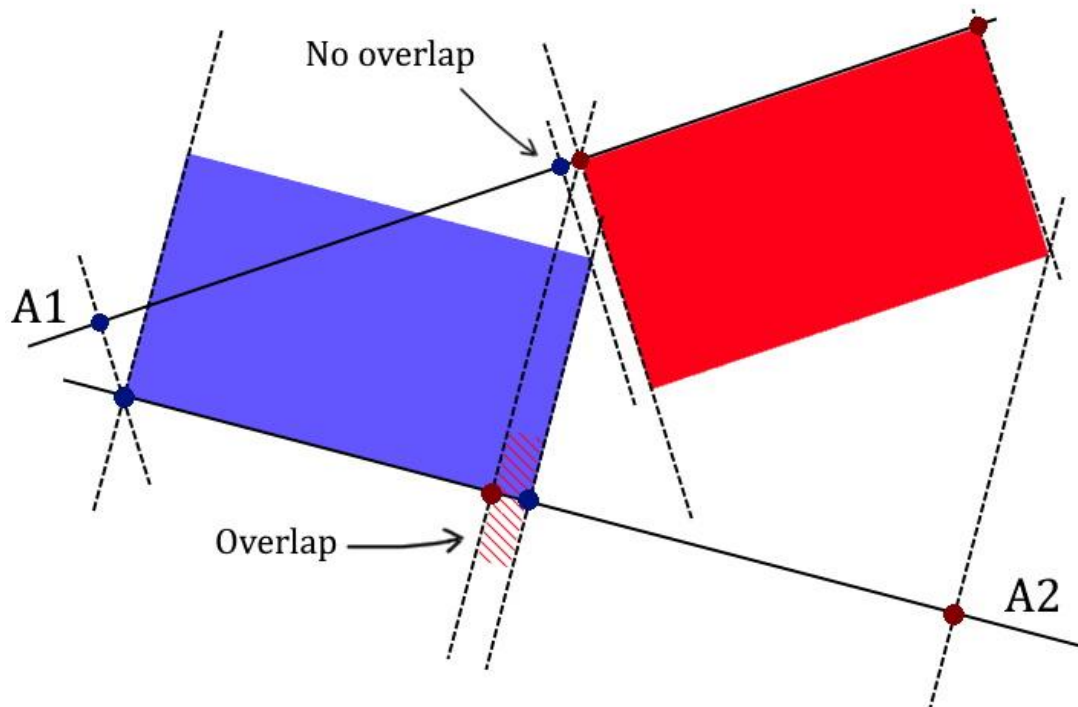
The collision verification method described above is applied for OBBs in a slightly more efficient way. Indeed, rectangles have a pair of parallel sides, and the overlap of corners along two parallel axes is the same – this saves from computing overlaps along half of the rectangles' sides. Also, the maximum and minimum of the projections of a rectangle's corners along one of its own sides are simply the corners of that side – which saves from checking the two other corners.

To make collision computations faster, a bounding circle was assigned to each piece. A bounding circle is a circle that encompasses all the piece's OBBs within its area. Bounding circles were determined manually in the piece characterisation tool detailed in Part 4.

A collision between two circles requires a comparison of the square of the sum of their radiuses to the square of their distance – which is easily determined using Pythagoras' theorem. This necessitates only a few operations, making collision computations for circles very efficient.

The collisions between two pieces are finally verified using the following method:

1) Check collision between the pieces' bounding circles. If these do not collide, the pieces do not collide.
2) If the bounding circles collide, check collisions between each one of the pieces' OBBs. If any of these collide, the pieces collide. Otherwise, the pieces do not collide.

Section 11.2 describes a test on the performance gained by means of adding bounding circles.

10.7 Avoiding repetitive computations

Once a basic version of the algorithm was implemented, some simple single-loop tracks could be generated. However, the algorithm would find tracks only for a very limited number of input pieces. Before making progress on obtaining multi-level and multi-loop tracks to generate, some improvements were necessary to improve performance.

Most sets of BRIO™ pieces contain multiples of the same pieces. For example, the "medium straight" or "large curve" pieces are very common and often picked several times. When a collision is detected, the next available piece is tried instead – however, when that piece has the same ID as the previously tested one, it is clearly pointless to attempt its placement, since the two pieces have the same geometry. To avoid repeating the same work multiple times, a set is created and made to contain the IDs of pieces that were unsuccessfully tested. Before attempting to place any piece, it is verified whether its ID is present in that set: if it is, the piece is skipped.

To get an idea of the impact this has on performance, one can consider the following formulae. The number of arrangements of N pieces (ignoring arrangements rendered impossible due to collisions) is the factorial of N. However, taking into account that a number M of pieces have the same ID in the set reduces the number of possible arrangements by the factorial of M.

For example, the number of arrangements of a set of 20 pieces is $20! \approx 2 * 10^{18}$. However, if half of these pieces are known to be of the E type, and the other half of the A type, the number of arrangements is drastically reduced: $\frac{20!}{2*10!} \approx 3 * 10^{11}$. This is nine orders of magnitude smaller than the previous number. This simple example shows that this improvement economizes a significant number of useless computations.

10.8 Further improvements on performance

The improvements detailed in Section 10.7 accelerated the program and allowed larger tracks to be generated. However, the time taken to generate a track with a certain set of input pieces was unpredictable and highly variable dependant on the seed chosen. For the same set of pieces, one seed could lead to the generation of a track in a few milliseconds, while another seed could require a generation time of more than 10 minutes.

The reason for this is that sometimes a piece got placed in such a way that generation was doomed to fail for a large number of attempts. For instance, one piece could be positioned in front of the Validation Connector and block it. Or a piece could be placed too far from the Validation Connector, making it impossible for any of the following pieces to reach it.

While these issues could be addressed individually by using heuristics, a different solution was found. Instead of starting a generation and waiting until it completes, the algorithm was modified to attempt generation only for a certain number of recursions. When this number is reached, the generation gets interrupted, all the pieces are taken off the board, and the generation restarts with a different initial order for the pieces.

It turns out that interrupting and restarting the generation after a small number of recursions, and attempting many generations in quick successions makes the program a lot more efficient, making generating times more consistent with varying seed numbers.

Measurements of efficiency gains obtained thanks to this addition are detailed in Section 11.2.

10.9 Multi-level tracks

There exist two kinds of ascending BRIO™ pieces [10]: the *N* piece, which essentially is an ascending version of the *D* piece, and the *N1* piece, a version of the *N* piece with two pin-type connectors instead of a pin and a hole. Only the *N* piece was added to the set of available pieces. This addition makes it possible for users to build multi-level tracks.

Only a small number of modifications were required for multi-level track generation to work. The following main modifications concerned collision-computation, piece-connection and track-validation functions:

- Collisions were made to be ignored when two OBBs were positioned on different levels.
- To connect a piece to the rest of the track, all the piece's components were first raised or lowered to the level of the track's open connector. A piece could be raised to any level, but it was important to check that none of its components were ever lower than ground floor.
- The validation conditions, detailed in Section 10.4, were slightly modified: an additional verification was implemented to make sure that the two end connectors were located at the same height.

The addition of ascending pieces brought a new kind of performance issue, concerning certain kinds of sets of input pieces. When an odd number of ascending pieces is picked, the ends of the track are always positioned on different levels, making it impossible to create a closed loop. To prevent this, any odd ascending piece systematically removed from the set of pieces before beginning track-generation.

10.10   Multi-loop tracks

Once single-loop tracks could be generated in decent amounts of time, and multi-level tracks could be implemented, the focus moved to multi-loop track layouts. The track-generating algorithm was modified substantially to enable this. A divide-and-conquer approach was used to tackle the problem of multi-loop track-generation.

The algorithm starts by determining the number *N* of loops that can be built for the given set of pieces – this is equal to the number of pairs of three-connector pieces available, plus one for the initial loop. Then, the set of pieces is partitioned in *N* subsets of equal sizes. It is ensured that exactly one pair of three-connector pieces are located in each subset, except for the last one. A loop is then generated for each subset in order, with the two 3-connector pieces positioned in a loop providing validation conditions for the next loop.

All the pieces are positioned in a single C++ vector, and the generation occurs "in place"; two indices are kept in memory, indicating the pieces that are available for constructing the current loop.

An outline of the algorithm for generating multi-loop tracks is as follows:

- function generateMultipleLoops(Piece[] selection) {
    - // Remove odd ascending piece and odd 3-connector piece
    - clean(selection)
    - // Get the number of 3-connector pieces in the set
    - int threeConPieces = countThreeConPieces(selection)
    - // Shuffle all the pieces together
    - shuffle(selection)
    - int numberLoops = 1 + threeConPieces / 2
    - // Locate indices of pieces available to construct the first loop
    - int indexStart = 1
    - int indexEnd = selection.length / numberLoops
    - // Place the first piece
    - selection[0].place()
    - for (int i = 0; i < numberLoops; i++) {
        - // Ensure that there are an even number of ascending and 3-connector pieces in the set
        - sanitise(selection, indexStart, indexEnd)
        - // Generate the loop
        - generateLoop(selection, indexStart, indexEnd)
        - // Position the indices for the next loop
        - indexStart = indexEnd
        - indexEnd = indexEnd + selection.length / numberLoops
    - }
- }

Note that this is a simplified version of the actual algorithm. For instance, the way to deal with the unsuccessful generation of a loop is not specified here. If a certain loop fails to get generated after a certain number of attempts, the generation falls back to the previous loop. After too many failed attempts at generating the first loop, the program is interrupted.

Examples of multi-loop tracks are shown in Figure 25, Figure 26 and Figure 27.
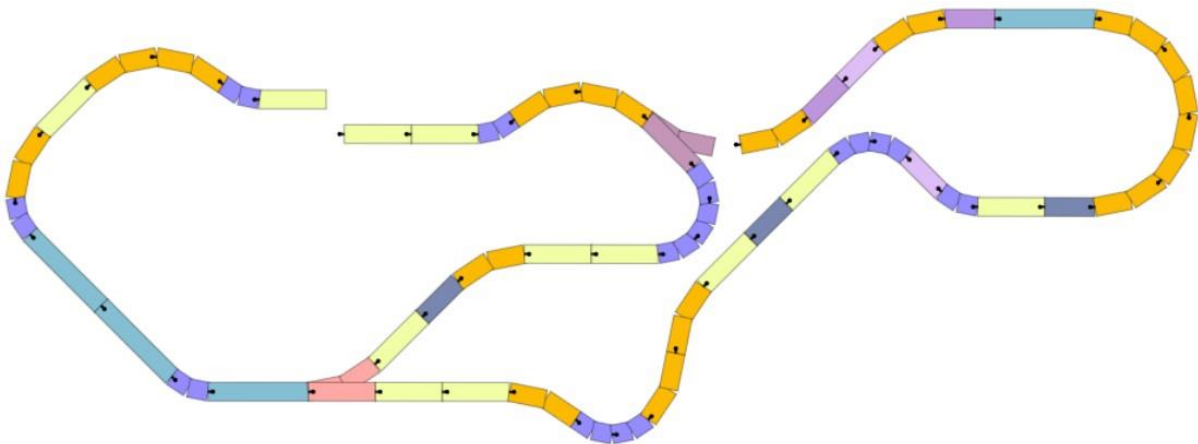
Figure 25: 2-loop track, generated using the set {15E, 12A, 4D, 3A1, 2B1, 2C1, 13E1, 1L, 1M}, close validation conditions, seed 2.

Figure 26: 2-loop and 2-level track, generated from the set {20E, 10A, 5D, 15A1, 30E1, 2L, 2M, 2N}, close validation conditions, seed 1. Pieces outlined in black lie on the ground, and those outlined in blue are at level 1.
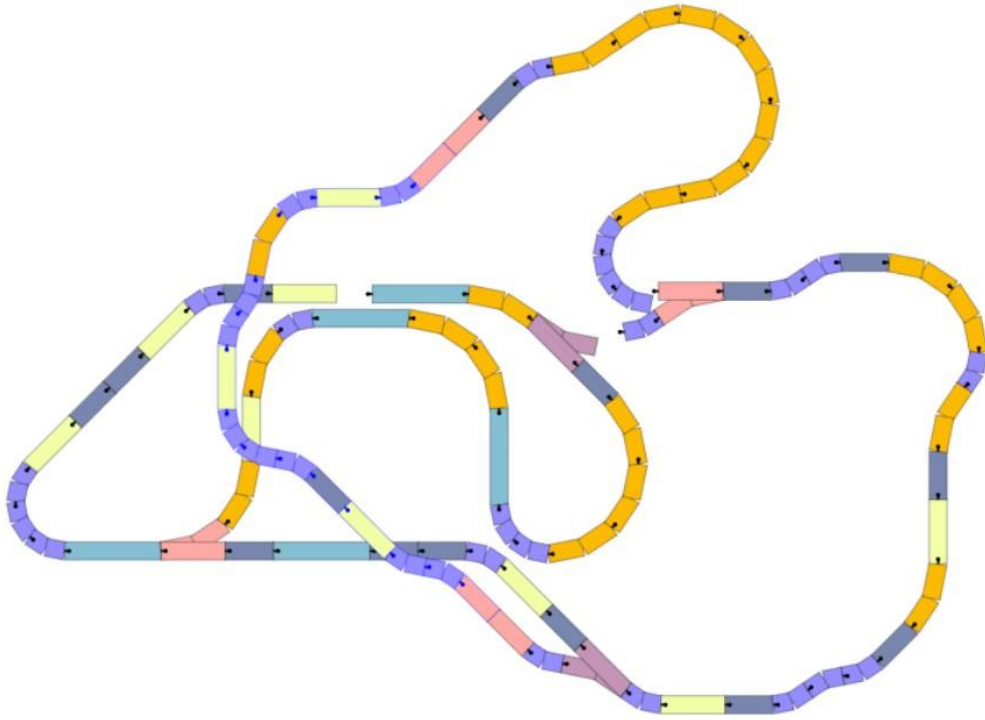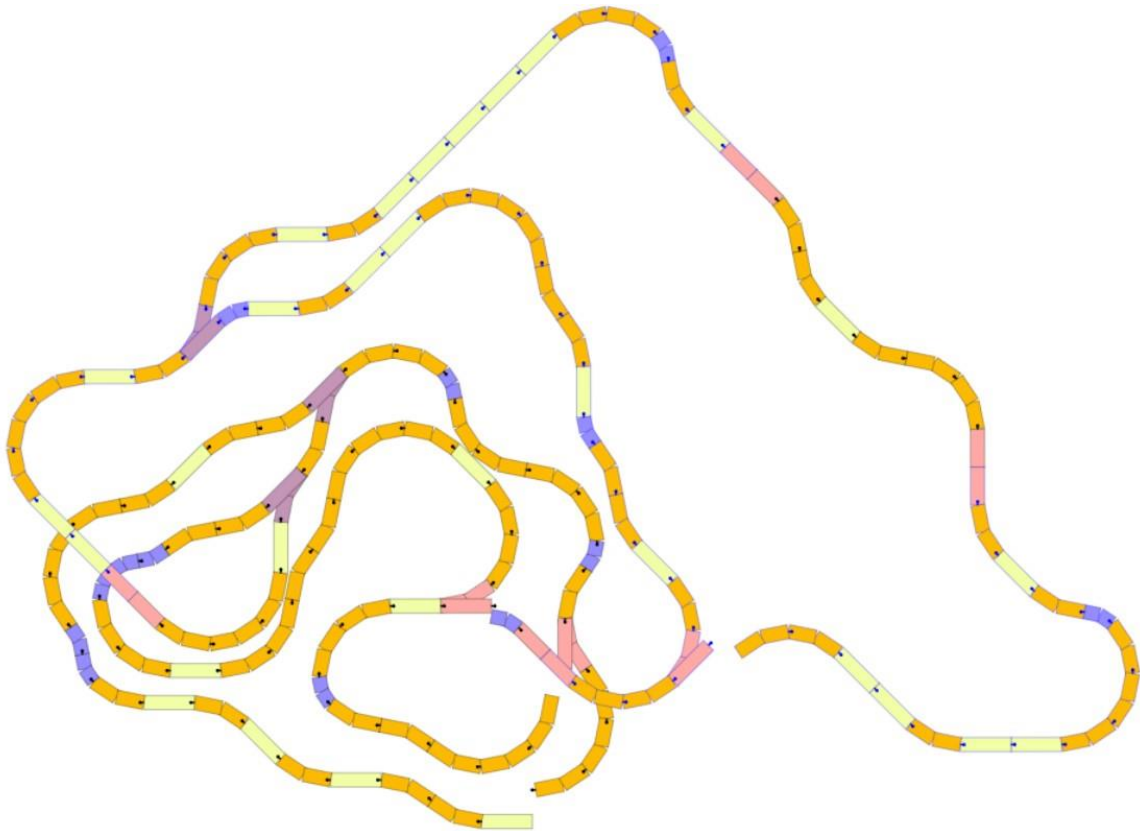

Figure 27: 3-loop and 2-level track, generated from the set {80E, 30A, 13E1, 3L, 3M, 4N}, close validation conditions, seed 1. Pieces outlined in black lie on the ground, those outlined in blue are at level 1.

10.11   Addition of a heuristic

Sections 10.6, 10.7 and 10.8 discuss improvements made in order to increase the efficiency of the generation of tracks, with the addition of bounding circles to cut down on the time taken to evaluate collisions. They also discussed how to prevent repeating the same computations for pieces of the same type, and the change of the algorithm to restart generation from scratch after a certain number of failed attempts. These alone make it possible to generate tracks from reasonably large initial sets of pieces within acceptable times.

A heuristic was added to further reduce track generation time. This heuristic is based on the following idea: a piece should never be placed further away from the Validation Connector than the total distance that can be covered by the remaining available pieces.

To implement this, it was necessary to determine the maximum distance covered by each type of piece. This is set to be equal to the Euclidian distance between the two connectors of the piece that are most distant from each other. This distance was computed for each piece type and stored in the "Pieces.json" file.

Piece placement is prevented if the distance from its open connector to the validation connector is larger than the total distance of the remaining pieces. Computing the total distance that can be covered using the available pieces is an O(N) operation. To avoid having to repeat this operation each time a new piece is placed, an *availableDist* property is added in order to retain the distance available at each step. When a piece is placed, its distance is removed from *availableDist*, and when a piece is taken back from the board, its distance is reinjected to *availableDist*. This makes checking the total distance of available pieces an O(1) operation.

10.12   Preventing impossible layouts

A problem arises owing to the implementation of the Vario System detailed in Section 10.4. A track is considered to be closed when its ends are close enough to each other. However, the ends do not necessarily touch each other, and due to this, the track can sometimes end up having pieces between its two ends. An example of this is shown in Figure 28.
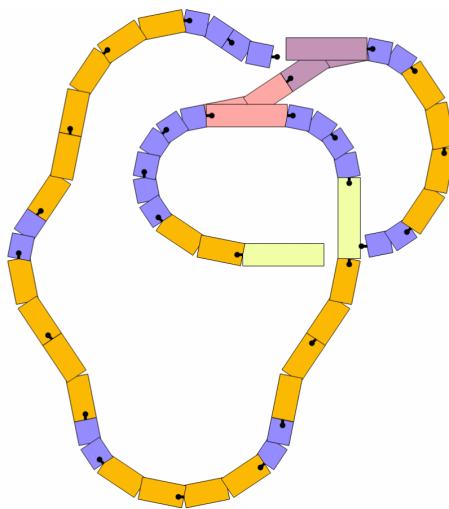

Figure 28: Track generated with pieces between its two ends, making it impossible to close.

To tackle this problem, an extra validation condition was created. This new condition consists in adding a thin OBB between the two ends of the track, and checking whether this OBB

collides with any of the placed pieces. The OBB is removed once collisions have been checked. If the OBB collides with any piece, then some pieces must lie between the track's ends, so the track is rejected.

To avoid too many tracks from getting rejected, the two end pieces are discounted from these collision verifications. Figure 29 shows an example of a track that would get rejected if the end pieces weren't ignored when looking for collisions with the OBB – this is undesirable, as the track shown should be possible to close through use of the Vario System.
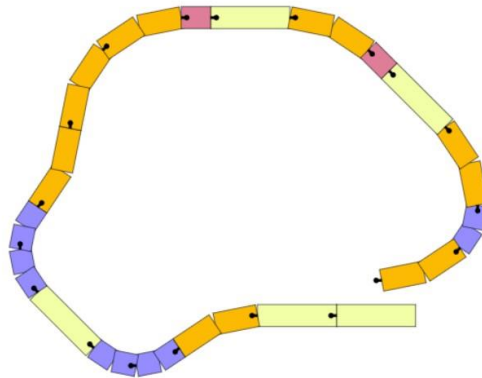
Figure 29: Example of track impossible to generate with a too strict version of the new validation condition.

This new validation condition successfully prevents impossible single loop tracks from getting generated. However, in slightly rare cases, certain impossible multi-loop tracks can still arise. Such examples may occur when a loop is generated with a small gap between its two initial connectors – pieces of the next loop may get placed through this gap without violating the newly created validation condition.

11 Evaluation

This section evaluates the implemented application. A comparison of the application with the initial objectives is given. Some tests are presented, evaluating the performance of the application. Then, a testing of the application for building a track with real BRIO™ pieces is discussed. Some of the application's current limitations are detailed and some ways in which those could be improved are also explored. Finally, the application is critically evaluated in comparison with related work.

11.1 Comparison with objectives

This section discusses the completion of the objectives determined in the DOER at the start of the project. These are detailed in Section 6.

Primary objectives

- *An interface should allow users to select track pieces from a list of basic pieces.*

This objective was completed, as discussed in Section 9.5.

- *An algorithm should generate one or multiple tracks that can be generated from a list of pieces. Whenever possible, the tracks generated should contain at least one closed loop.*

Single-loop track generation was made possible, as discussed in Section 10.2. It was made impossible to generate tracks that contained no closed loops.

- *A 2D interface should be available to the user to view the tracks generated.*

This objective was completed, as discussed in Section 9.2.

- *The tracks should be generated in a reasonable time.*

Optimisation of the track was a main concern throughout the project. Sections 10.6, 10.7, 10.8 and 10.11 discuss the measures taken to improve track-generation performance. The application is now able to generate single-loop tracks in acceptably short times for relatively large sets of pieces. Performance falls however for large numbers of loops – it is difficult to obtain tracks generated with three or more loops at optimised generation rates.

Secondary objectives

- *The dimensions of the room should be specifiable by the user. The train tracks generated should then fit in the room specified.*

This objective was not completed – a discussion on this can be found in Section 11.3.

- *Ascending pieces should be added to the list of available pieces. This would allow the generation of multi-level train tracks.*

Ascending pieces were added to the list of available pieces, and made it possible to construct multi-level track layouts. This is detailed in Section 10.9.

- *Track pieces with more than two connectors should be added to the list of available tracks.*

This objective was completed, making it possible to generate multi-loop layouts. The implementation of this is detailed in Section 10.10.

- *Track pieces with a single connector should be added to the list of available pieces.*

This objective was not completed. It was decided to concentrate only on generating layouts containing closed loops, which explains why single-connector pieces were not added to the list of available pieces.

- *The connection between any two pieces is not perfect – this allows for the closing of tracks for which the extremities are "close enough" to each other. This should be encoded as part of the track generating algorithm.*

The requirement to include an implementation of the Vario System turned out to be essential for generating the majority of track layouts – without it, constraints are too strong and very few tracks can be found. A discussion of the Vario System and its implementation in the algorithm can be found in Section 10.4.

Additional objectives

- *More sophisticated pieces could be added to the list of available tracks.*

This objective was not fulfilled. However, more complicated pieces such as tunnels could be determined by slightly extending the piece-characterisation tool detailed in Section 8. This could be done by making it possible to specify arbitrary OBBs for certain pieces. The track-generating program should be able to include such pieces in generated tracks without modifications to the track-generating algorithm, as long as such pieces contained two or three connectors.

- *The position of supports to hold tracks which aren't on ground level should be specified to the user.*

This objective was not completed. One way in which this problem could be tackled would be by representing such supports as OBBs, and these could be added to pieces that are not on the ground.

- *Track generation time could be decreased by making use of multi-threading if possible.*

This possibility was closely considered. However, multithreading in a program compiled to WebAssembly is a relatively new feature that only Chrome and Firefox appear to possess [21]. Also, certain special server headers – the COOP and COEP headers – must be specified to allow multithreading in those browsers. Because of low browser support, multithreading was not used in this program.

- *The use of Web Assembly to run the track generating algorithm more efficiently on the frontend of the application could be considered.*

As discussed in Section 10.1, the C++ program was compiled to WebAssembly in order to make the web-application infinitely scalable by getting rid of any interaction with a server.

- *The user should be able to select the number of loops the tracks generated should contain.*

This was not directly implemented – instead, users can choose how many loops to generate by changing the number of pairs of switch pieces.

11.2 Performance tests

Throughout the project, multiple methods were employed to improve the performance of the program and make it possible to generate tracks with larger and more complex sets of pieces. This section details some tests that were made to measure the impact these additions had on performance.

Bounding circles – see Section 10.6

An estimate of the performance gained through addition of bounding circles was calculated using the data presented in Figure 30. This data was obtained by timing the generation of a track with a range of seeds. Due to the way in which seeds are implemented, the addition of bounding circles, while changing the collision computation time, does not affect the way in which the track is generated for a specific seed. This allows for easy comparison between the track generation with and without the addition of bounding circles.

From this data, one can see that the track generation with the inclusion of bounding circles was more efficient than the one without bounding circles, for every single seed tested. For this set of pieces, the addition of bounding circles decreased the average track-generation time by a factor of almost 7, which is a very substantial performance gain.
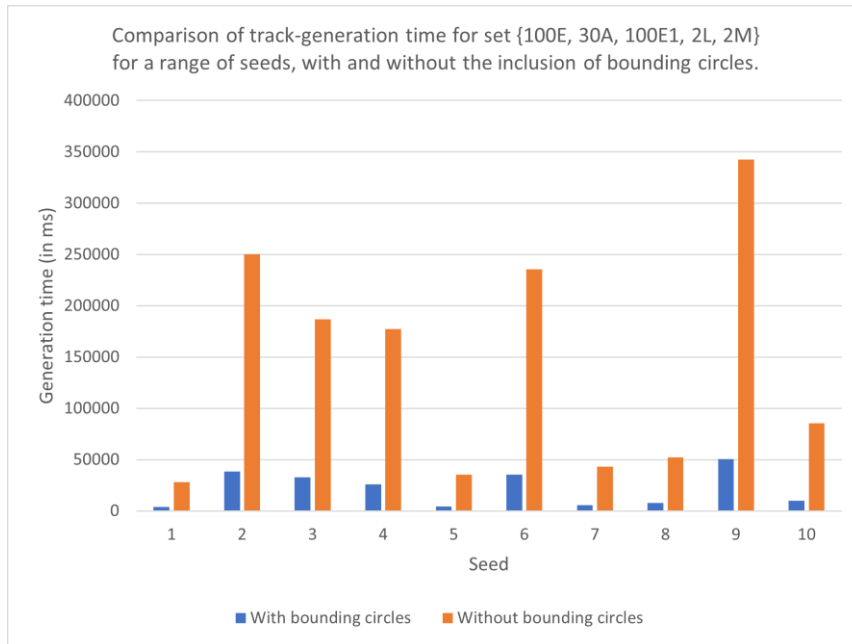

Figure 30: Comparison of time taken for generating a track, with and without the inclusion of bounding circles.

Prevention of repetitive computations for pieces with the same ID – see Section 10.7

Figure 31 shows measurements made on the time taken to generate a track with the set of pieces {20E, 20A, 10A2, 10E1}, for seeds ranging from 1 to 10. *Series 1* corresponds to the case where collision computations for pieces with the same ID are prevented. *Series 2* corresponds to the more naïve implementation of the algorithm, where all pieces are checked independently of their ID.
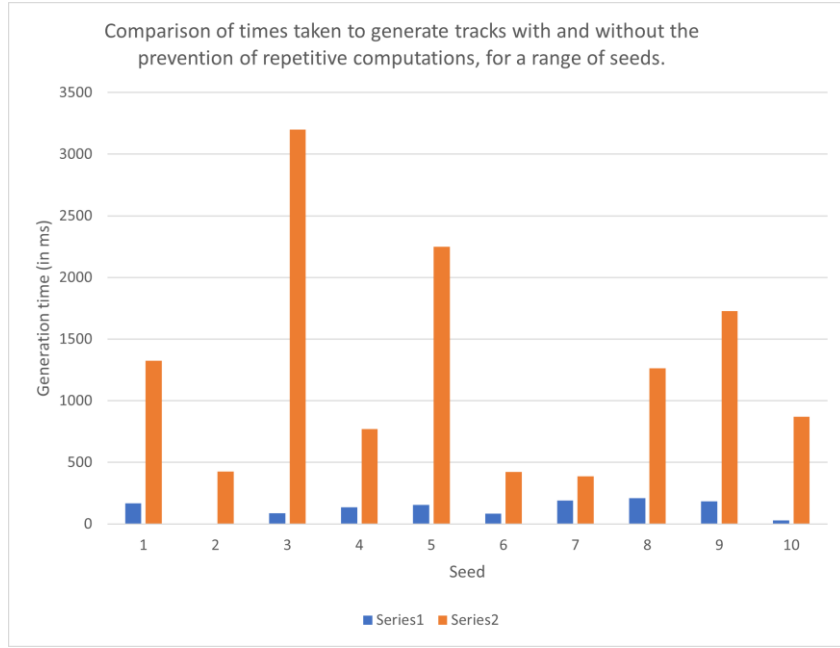
Figure 31: Bar chart comparing the times taken to generate a track with and without the prevention of same-piece computations.

Generation was always faster in *series 1*. On average, it took 124 milliseconds to generate a track in *series 1*, versus 1.264 seconds in *series 2*. This means that it took on average $1264/124 \approx 10$ times longer to generate a track without the improvement in place.

The same experiment was repeated but with twice the number of pieces - the set now was {40E, 40A, 20A2, 20E1}. Results are shown in Figure 32.
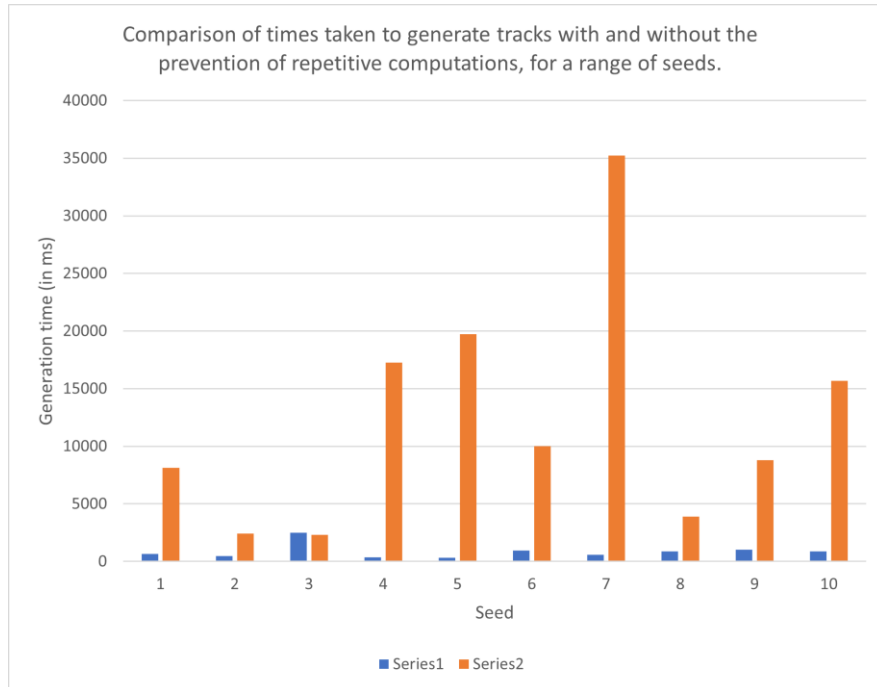

Figure 32: Bar chart comparing the times taken to generate a track with and without the prevention of same-piece computations, with twice the number of pieces as in the previous experiment.

This time, the average time required to generate a track was 859 milliseconds in *series 1* and 12,344 milliseconds in *series 2* – the factor between the two now being $12344/859 \approx 14$. The impact this improvement has on performance grows with larger sets of pieces. This is expected, because the number of possible piece arrangements grows with the factorial of the number of input pieces – see Section 10.7.

Stopping and restarting generation in quick successions – see Section 10.8

The test is the following: generation is attempted for the simple set of pieces {30E, 30A}, with close validation conditions, for seeds ranging from 1 to 10. This is repeated twice: *series 1* corresponds to the program attempting a single long generation, and *series 2* is the same test but with the inclusion of the improvement. Figure 33 shows these results in a bar graph, with generation times presented on a logarithmic scale. In the first case, the program was allowed to test one-billion piece placements, after which it was forced to stop. In the second case, the program restarted a new generation from scratch after 10 000 failed piece placements.
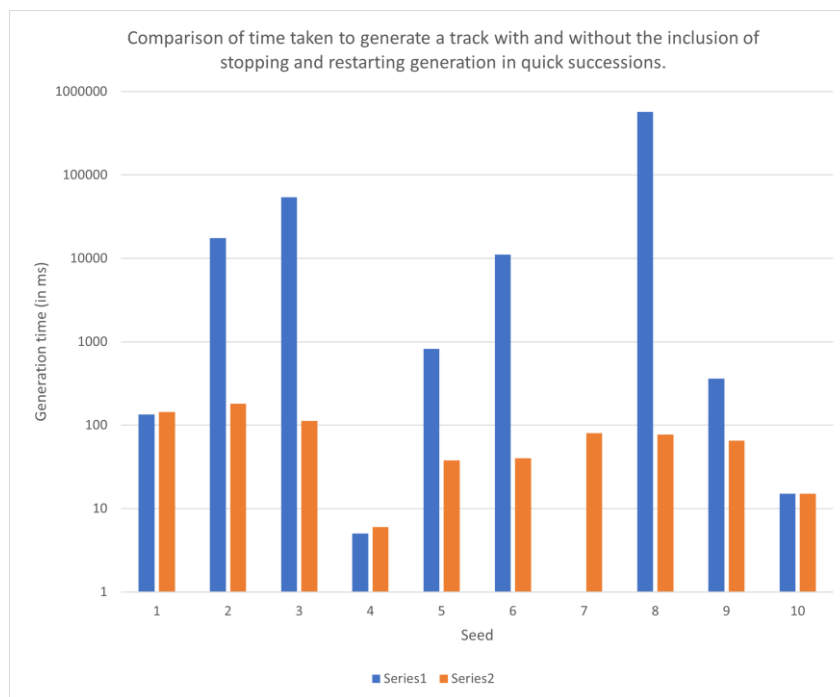

Figure 33: Time taken to generate a track for series 1 and 2, for seeds ranging from 1 to 10. Generation times are presented on a logarithmic scale.

In *case 1*, the generation times were very uneven: generation was completed in less than 1 second in five cases, took more than 10 seconds in three cases, and more than 9 minutes for seed 8. For seed 7, the one-billion piece placement limit was reached before any track could be found. Ignoring the generation for seed 7, the average track generation time for *series 1* was around 82 seconds. In *series 2*, tracks were found in less than 0.2 seconds for all seeds, with an average generation time of 76 milliseconds. Ignoring the failed generation in seed 7 in *series 1*, the generation in *series 2* was on average $81971/76 \approx 1078$ times faster than in *series 1*.

This simple test is sufficient to show the massive performance improvement brought by the method of stopping and restarting generation in quick succession.
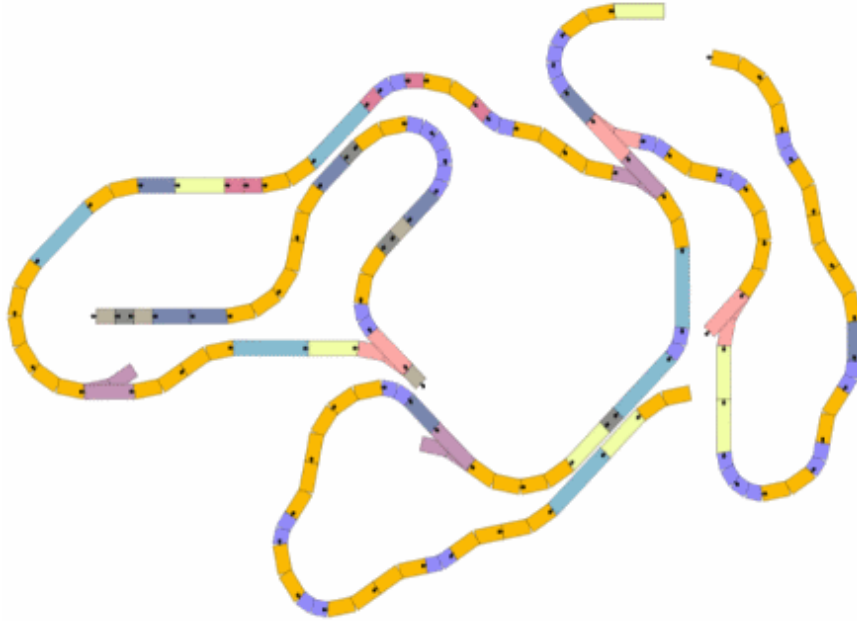
11.3 Testing with real BRIO™ pieces

Figure 34: The track Tom generated.

Figure 35: Tom's track built with real BRIO™ pieces.

Tom agreed to test the application and build a real BRIO™ layout from one that he generated using the application with the set of pieces he owns. Figure 34 shows the track that Tom generated using the program, and Figure 35 shows the layout he constructed with real BRIO™ pieces. The close similarity between the generated and the real track layouts shows that the pieces were determined with correct dimensions.

Tom discovered a few points on which the application could be improved and provided some useful feedback from this test. Some of Tom's points relate to improvements to the user interface:

- The correspondence between piece type and colour is not displayed on the interface, making it hard at first to figure out which piece is which on the generated output. This could be fixed by adding a simple table displaying these correspondences.
- There is no option to specify a piece as "placed", which makes it difficult to keep track of which parts of the track have already been built. The possibility to click on a piece to make its colour fade, marking it as "placed", would fix this issue.
- The seed used for the generation of the track was not displayed at the time, making it impossible to re-create a track previously generated. This was fixed after Tom's testing, as detailed in Section 9.7.
- The pieces specified to generate a certain track were lost when the web page was refreshed. This was fixed by adding piece numbers specified in user inputs saved to the page's local storage, as discussed in Section 9.7.

Some of Tom's findings concern the track-generating algorithm itself:

- There is no way of specifying a constraint on the size of the room, making it difficult to find tracks that can fit into a certain room. Given more time to work on the project, this problem could be addressed in the following way. By making it possible to specify a width and a height, a room could have been represented as a large rectangle, and any attempt at placing a piece outside of this rectangle would be made to fail.
- Some parts of the track Tom generated contained redundant piece sequences. Examples of such placements are shown in Figure 36 and Figure 37. It would be hard to modify the algorithm to prevent such redundant sequences, and any attempt at fixing the issue would likely have a negative impact on the program's performance. The current implementation of the program is likely to generate such redundant placements, but users are welcome to build a slightly different version of the track they obtained avoid such issues.
- The program struggled to generate tracks with Tom's set of pieces, timing out for many different seeds. The author believes the reason for this is that the program always attempts to generate a track with the maximum number of loops possible. Tom's set of pieces contained 7 switch pieces, one of which got automatically thrown away at the start of the generation, as explained in Section 10.10. Using the 6 remaining switches, a maximum number of 4 loops can be generated. The program usually has trouble generating tracks with a high number of loops, especially for relatively small sets of pieces, which is why it struggled to make the generation work. Some arguably better tracks could have been built from Tom's pieces by removing some of the switches to create less loops. Figure 38 and Figure 39 show examples of 2-loop and 3-loop tracks that can be generated with Tom's pieces. One way of addressing this issue could be to make the program attempt generation with the maximum number of loops first, and if it fails, successively try generation with less loops until a track is found.
- Tom's track was generated with an incorrect loop, with pieces located between two end connectors. This issue was later addressed, as discussed in Section 10.12.

Figure 36: Redundant piece placement – the succession of a two-hole piece and a two-pin piece could be replaced by a single straight piece. Picture taken by Tom.

Figure 37: Redundant piece placement – the succession of straight pieces of different sizes could be replaced by a single longer straight piece. Image taken by Tom.
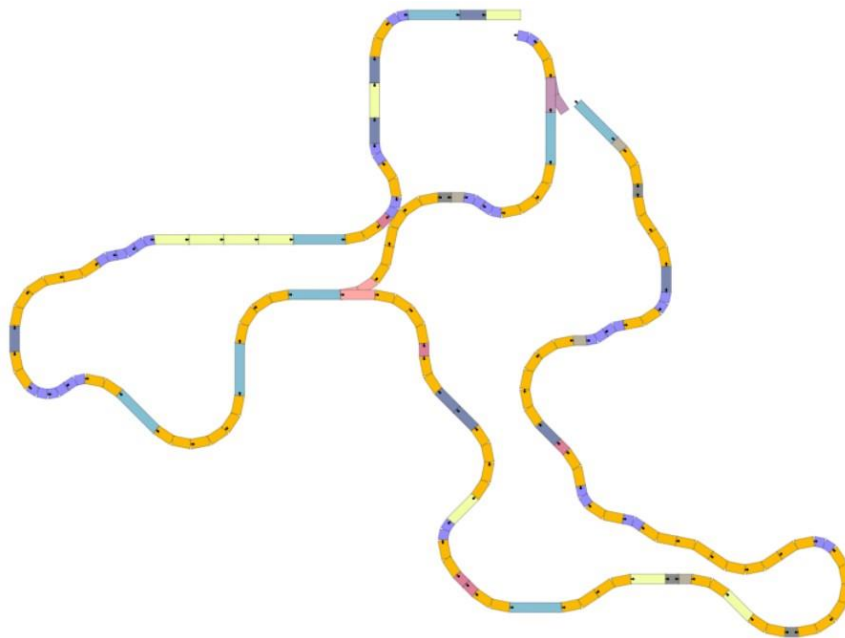

Figure 38: Example of a 2-loop track layout that can be obtained using Tom's set of pieces, with close validation conditions. This track was generated in just under 11 seconds.
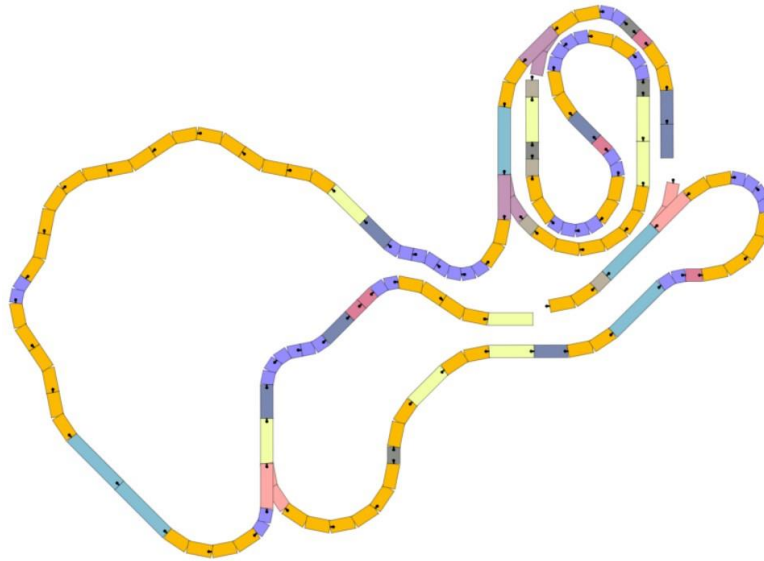
Figure 39: Example of a 3-loop track which can be obtained using Tom's set of pieces, with close validation conditions. This track was generated in just under 38 seconds.

11.4 Limitations and possible improvements

Following on from Section 11.3, this section highlights more of the application's current limitations and explains how these could be improved, given more time to work on the project.

One of the things that slows down the current track-generating algorithm is the verification of collisions. While efforts were made for collisions to get computed faster (see Section 10.6), the fact that collisions with all the pieces placed need to be computed for every newly placed piece has a strong negative impact on performance. A way in which this issue could be addressed would be by imposing a restriction on the size of the room, instead of allowing pieces to be positioned anywhere in an infinite 2D space. The space in which pieces can be positioned could be a rectangle of fixed size, and the area within this rectangle could be partitioned into a grid of small square cells. Each piece would be assigned the cell on which it is positioned. Then, upon arrival of a new piece on the board, collisions would be checked not with all the pieces, but only with those lying on nearby cells. Of course, certain edge cases would need to be considered, one of which would be that certain pieces could lie on intersections between cells, but this could be relatively easily solved by assigning multiple cells to those pieces. Such an improvement would reduce the complexity of calculating collisions with previously placed pieces – if there are N pieces on the board, the complexity of calculating collisions with the current method is $O(N)$, but could be lowered to $O(1)$ using this new approach.

Another way in which the generation of tracks could be made more efficient would be by implementing a heuristic. This heuristic could come forth once a certain proportion of the pieces have been placed and would force the next pieces to gear towards the validation connector. It is not totally clear how this would be implemented best, but a simple version could be to systematically choose the orientation of curved pieces that steers the track closer to the validation connector. The author believes that the addition of this new heuristic, together with the partitioning of space to improve collisions, could make track generation a lot more efficient and enable generation of larger tracks.

45

As discussed in Section 10.12, the current implementation of the track-generating algorithm makes it possible, in some specific cases, to generate tracks with an interrupted loop. An extra OBB is placed between the ends of the track, and collisions between placed pieces and this OBB are verified to check whether any piece interrupts the track. Issues might arise for multi-loop tracks, where pieces may end up being placed between the two ends of a previously constructed loop. This problem could be prevented relatively simply – instead of discarding the extra OBBs, these could be kept in a list. Every time a new piece is placed, collisions against the OBBs in that list would be checked, and this would prevent any piece from being positioned in such a way that it interrupts a previous loop.

11.5 Related work

The article 'Making Racing Fun Through Player Modeling and Track Evolution' by Togelius et al. features elements which may be compared to those presented in this project. Togelius et al. present a method for automatically generating tracks "tailor-made to maximize the enjoyment of individual players in a simple car racing game" [22]. The profile of a human player is analysed based on the latter's performance on different handcrafted test tracks. A track is then encoded using a sequence of fixed-length fragments. Each fragment is either a straight segment or a curve with one of three possible curvatures. Fragments can also contain obstacles. The procedural generation is based on an evolutionary algorithm that uses parameters determined from the analysis of players' profiles, to modify the segments of a pre-determined track with different probabilities.

These fixed, discrete fragments are comparable to the BRIO™ pieces used to generate track layouts. However, the recursive method for generating BRIO™ tracks is very different from the procedure presented in this article. This can be explained by the diverging objectives of the procedural generation algorithms. On the one hand, the BRIO™ track generation has the constraints of the user-specified set of pieces and has the requirement of generating closed loops. On the other hand, the procedural generation presented in this article does not require generating closed loops, neither does it prevent tracks from turning back on themselves, leading to overlaps. It instead focuses on the creation of racing track layouts that present an interesting challenge for specific players.

Unlike the procedural generation of racing tracks presented in this article, the author's method for generating BRIO™ train tracks does not give users any particular guarantees on the quality of tracks generated. However, a significant asset of the BRIO™ track generator lies in its capacity to find track layouts under strict constraints.

12 Conclusion

The application developed by the author allows users to obtain automatically generated BRIO™ track layouts. Fifteen types of pieces were implemented and made available for users to choose from. An interface allows users to select pieces and visualise generated tracks. Track layouts of potential strong interest to BRIO™ enthusiasts can be obtained, thanks to the addition of multi-level and multi-loop track-generation. The generation was also made efficient for inputs containing large numbers of pieces.

However, some challenges remain. The possibility for example of making generated tracks fit in a rectangular room with dimensions specified by the user is one improvement that could be implemented. Another challenge could be to adapt track generation to rooms of different shapes. But that is another story!

13 User manual

13.1 Starting the application

The program can be run by starting a local development server in the "docs" folder. Indications on how to re-compile the C++ program to WebAssembly can be found in the README.md file.

Alternatively, an online version of the program is running on a static website hosted on Github Pages, accessible using the following URL: https://alex-kings.github.io/brio-project/.

13.2 Generating a track layout

To generate a track layout, a set of pieces must be selected. A minimum of eight curved pieces needs to be specified for any track to be generated – less curved pieces do not lead to closable loops.

As an example, basic tracks can be generated using a set with 12 large curves and 10 medium straight pieces.

Tracks generated can be visualised on the right side of the user interface. The track can be moved around by clicking and dragging the mouse on the canvas. It is possible to zoom in or out by scrolling. Using a physical mouse to move and zoom on the canvas is highly preferable to a touchpad.

14 Acknowledgements

15 Ethics form

Title of project

BRIO track generator
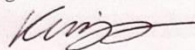
Name of researcher(s)

Alexandre Kings

Name of supervisor

Tom Spink

Self audit has been conducted **YES** ☒ **NO** ☐

This project is covered by the ethical application CS15727.

Signature Student or Researcher

Print Name

Alexandre KINGS

Date

23/09/2022

Signature Lead Researcher or Supervisor

Print Name

TOM SPINK

Date

23/09/2022

16 References

[1] "Brio (company)," [Online]. Available: https://en.wikipedia.org/wiki/Brio_(company). [Accessed 18 March 2023].

[2] I. M. Joan Espasa Arxer, *Procedural Content Generation I,* 2023.

[3] "Minecraft," [Online]. Available: https://en.wikipedia.org/wiki/Minecraft. [Accessed 18 March 2023].

[4] "No Man's Sky," [Online]. Available: https://en.wikipedia.org/wiki/No_Man%27s_Sky. [Accessed 18 March 2023].

[5] J. T. M. J. N. Noor Shaker, "Chapter 2: The search-based approach," in *Procedural Content Generation in Games*, Springer, 2016, pp. 20-22.

[6] R. D. N. S. L. J. Togelius, "Towards automatic personalised content creation for racing games," *IEEE Symposium on Computational Intelligence and Games,* pp. 252-259, 2007.

[7] "2-D Layout Modeling: SketchUp," [Online]. Available: http://woodenrailway.info. [Accessed 28 January 2023].

[8] "Hyperplane separation theorem," [Online]. Available: https://en.wikipedia.org/wiki/Hyperplane_separation_theorem. [Accessed 24 March 2023].

[9] "Oriented Bounding Box," [Online]. Available: https://subscription.packtpub.com/book/game-development/9781787123663/7/ch07lvl1sec70/oriented-bounding-box. [Accessed 26 March 2023].

[10] "BRIO Track Guide," [Online]. Available: https://woodenrailway.info/track/brio-track-guide. [Accessed 28 January 2023].

[11] "Canvas API," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. [Accessed 26 March 2023].

[12] A. Riškus, "Approximation of a cubic Bezier curve by circular arcs and vice versa," *ISSN 1392 – 124X, Information Technology and Control, Vol. 35, No.4,* 2006.

[13] BRIO, "Tunnel," [Online]. Available: https://www.brio.co.uk/en-GB/products/brio-world/buildings-tunnels-bridges/tunnel-63373500. [Accessed 30 January 2023].

[14] "Painter's algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Painter%27s_algorithm. [Accessed 24 March 2023].

[15] A. Shpakovsky, "Javascript canvas drag-and-zoom library," [Online]. Available: http://alexey.shpakovsky.ru/en/javascript-canvas-drag-and-zoom-library.html. [Accessed 21 January 2023].

[16] "WebAssembly," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly. [Accessed 25 March 2023].

[17] "Emscripten Documentation," [Online]. Available: https://emscripten.org/docs/. [Accessed 25 March 2023].

[18] "The Meson Build System," [Online]. Available: https://mesonbuild.com/. [Accessed 20 March 2023].

[19] "JsonCpp," [Online]. Available: https://github.com/open-source-parsers/jsoncpp. [Accessed 20 March 2023].

[20] "std::default_random_engine," [Online]. Available: https://cplusplus.com/reference/random/default_random_engine/. [Accessed 24 March 2023].

[21] I. Stepanyan, "Using WebAssembly threads from C, C++ and Rust," [Online]. Available: https://web.dev/webassembly-threads/. [Accessed 25 March 2023].

[22] J. T. R. D. N. S. Lucas, "Making Racing Fun Through Player Modeling," 2006.