Master of Science in Informatics at Grenoble (MoSIG)
Master Mathématiques Informatique - spécialité Informatique
Option: Ubiquitous Interactive Systems

Novel Approaches for Real-time Assessment and Automatic Generation of Game Content

Mohammad Shaker

June 26, 2015

Research project performed at:
Center for Computer Games Research, ITU of Copenhagen, Denmark

Under the supervision of:
Dr. Noor Shaker, ITU of Copenhagen, Denmark

External reviewer:
Prof. Paolo Burelli, Aalbrog University Copenhagen, Denmark

Defended before a jury composed of:

[June]   2015

Abstract

In this thesis, we propose two novel approaches for real-time assessment of generated content and for rapid Content Generation (CG). More specifically, we present two approaches:

- The first is the **Projection-based Approach** that supports game designers by providing visual feedback about the space of interactions. The approach propose using the area of influence (IA) as a building block to tackle the problem. The IA, which identifies the possible space of interaction, is identified for each component given its physical properties. Then the IAs over all components are merged considering the components' type and context information to represent the final reachable, active space of the level. We've further extended the approach using it for content generation. We use a clone of Cut the Rope, the famous physics-based game, to serve as a testbed for this approach.
- The second approach is what we call the **Progressive Approach**. This approach combines the advantages of constructive and search-based approaches, thus providing a fast, flexible and reliable way of generating diverse content of high quality. CG is seen from a new perspective which differentiates between two main aspects of the gameplay experience, namely the order of the in-game interactions and the associated level design. The framework first generates timelines following the search-based paradigm. Timelines are game-independent and they reflect the rhythmic feel of the levels. The progressive, constructive-based approach is then implemented to evaluate timelines by mapping them into level designs. We've shown the approach's potential by applying it for the generation of content of physics-based puzzle games; Cut the Rope. We've further applied it for the generation of content of a different game genre; the 2D swipe puzzle games in NEXT and iNversion.

The results for both approaches in terms of performance, expressivity and controllability are characterised and discussed.

Contents

— 1 —
Introduction

1.1 Intro and Research Questions

Level design is an essential part of the game design process. Level designers put considerable time and effort into creating an interesting, playable level. This process is typically iterative where designers start with a simple sketch, check whether it is playable in its raw form and which parts work and don't work, and go back to the editing mode to change and embellish the level. This process is repeated until the designer is satisfied with the result.

In this study we introduce a new real-time approach to check for playability for designers of physics-based games, an area, to our knowledge, is still untapped. The new approach tries to answer some of the questions: What are the playability constraints in physics-based games? What are the commonalities of properties between physics-based games? Can a *real-time* system be implemented for checking for playability? Is the system *generic* enough to be applied in multiple physics-based games? Is it efficient? Can it be used for content generation? How it fares against other state-of-the-art approaches?

In the second part of the thesis we shift our focus to content generation. More specifically, we focus on creating a generic approach for content generation. We ask the questions: Can a generic approach be implemented for content generation regardless of the game genre? What are the compromises? Are there any downturns or any constraints? How efficient is it? Is it really *generic*? Can it be applied for the generation of content for games of different types? How it fares against other approaches for generating content in a specific game?

We try to answer some of these questions by deriving a new novel approach by combining two distinct approaches in procedural content generation for games, namely, the constructive, generate-and-test and search-based approaches [1]. We call it the *Progressive Approach* for content generation.

1.2 Motivation

Driven by the research questions, in the first part of this study, we to introduce a new version of an authoring tool for physics-based game, and in particular, we introduce a novel and fast method for generating feedback to game level designers during the design process. We use the popular physics-based puzzle game *Cut the Rope* as a testbed for our approach. We further show the applicability of this approach for the generation of playable content. To our knowledge, no other work is found in this area and this approach can greatly contribute to real-time authoring tools for games in general and physics-based puzzle games in particular. We discuss this in Chapter 4.

In the second part, we present a new attempt at combining the diversity, flexibility and playability-preserving ability of search-based approaches with the speed of constructive approaches for content generation. The result is a fast, efficient and generic approach for content generation in games. We call this a *progressive* approach to the generation of playable content. We show this approach applicability by showcasing it for the generation of content for physics-based game; Cut the Rope. We then give a glimpse of our future work derived by this study by applying the progressive approach for the generation of content for a different game in a different game genre; NEXT in the 2D Swipe games genre, a completely different genre than that of Cut the Rope. This is discussed in Chapter 5.

— 2 —
State of the Art

In the first part of this study we mainly talk about game design. Game design is complex and labour-intensive. Therefore, game developers have devoted large efforts to the development of reusable systems to facilitate game design and production. Third-party middleware companies also focus on reusability through providing essential game-independent functionalities such as path finding, animation and physics simulation [2, 3]. Most of these tools are designed for experts in game design and development.

Like for other labour-intensive processes, artificial intelligence techniques could be leveraged to assist and offload the designer. In particular, intelligent authoring tools could ease the design process and decrease time consumption, freeing the designer resource up for higher-level design tasks. Such tools should be easily accessible to game designers and provide comprehensible yet informative feedback [4, 5]. There has recently also been much interest in developing tools that automatically check for playability, e.g. through the use of AI agents designed specifically to learn the rules of the game and explore the possible playable space afforded by the design [5].

Also, there has been increasing interest in academia in the creation of authoring and mixed-initiative tools. Some notable works include *SkectchaWorld* a tool for modelling 3D world through the use of a simple visual interface [6], *Tanagra* a mixed-initiative design tool for 2D platformer in which human and computer collaborate to design game levels [4], *Sentient Sketchbook* a tool that aids the design of game maps and automates playability check [7], *Ropossum* a mixed-initiative design tool that permits automatic generation and testing of physics-based puzzle games [8, 9], and the assisted level design tool for the game *Treefrog Treasure* that enables visualisation of the reachability between the game components [10].

One notable example of an industry-developed game-specific authoring tool is the one designed for the game *Rayman Legends* [11] which provides a user-friendly interface supported by automatic adjustment of basic features such as lighting, colour, and basic platform. The tool also allows real time editing and adjustment of the different game artefacts.

One of the core aspects when designing game levels and when implementing a designer-assistance tool is playability. Playability checking could be done in different ways. There are more or less direct approaches, based on e.g. verifying the existence of paths between different points [12], measuring gap widths and drop heights [4] or proving the existence of solutions when the game mechanics or some approximation thereof can be encoded as first-order logic [13]. However, in many cases you need to actually play the level in order to show that it is playable, and thus requires us to build an agent capable of playing the game [14]. Building an agent capable of proficiently playing a given game is rarely straightforward. The results obtained by most agents are usually the path followed to solve the level, if the level is playable [9] or simply a negative feedback indicating an unplayable design. Attention is rarely given to analysing or providing more informative feedback about the full interactive space afforded by the design artefacts. This can be highly beneficial for game designers since it provides information about the unused game components and the playable space in the level.

This study introduces a new version of an authoring tool for physics-based game, and in particular a fast method for generating feedback to game level designers during the design process. We use the popular physics-based puzzle game *Cut the Rope* as a testbed for our approach. The AI inspects the current state of the game and projects possible future states; in contrast to the method used in a previous version of the tool, which was based on search in a constrained space of game actions, the new method is based on search in a a space of sequences of geometric operations. Part of the result of the method is visualisation of the playable area, i.e. the proportion of the game level in which actual interaction with the player can take place, on the game canvas. The area is plotted on the canvas and updated with every edit by the designer. Thus, the tool permits realtime visual feedback about the interaction space and whether the game is solvable. This is achieved by defining an *influence area* for each game component considering their physical properties and relative position. The final interactive area is the combination of the influence areas of all components presented in a design. Through accounting for the context information and the physical properties, the agent is able to accurately infer the playable space without the need to simulate the game.

The current work builds on previous work by [15] on procedurally generating content for the game using a search-based approach. This work was further extended in [9] by implementing a simulation-based approach for playability testing. The simulation used a Prolog-based agent that plays through the level and considers only the most sensible moves at each state. Although the agent is able to accurately detect a playable design, the method followed suffers from a relatively high processing time (checking a single level takes an average of 29.87 ± 58.28 sec). Furthermore, the agent stops searching after finding the first path that solves the level. Extracting all possible playable trajectories would yield a substantial increase in the processing time.

The purpose of the work presented in this study (in Chapter 4) is not only to identify the trajectory followed to solve the game, but also to provide an easy to interpret visual illustration of all possible playable paths in *real time*. We believe that this information is vitally important to game designers who are keen to easily analyse their design and to rapidly realise the impact of their design choices on player experience. This is discussed in Chapter 4.

In the second part of the study, we mainly talk about Content Generation, specifically, a *generic* approach for content generation. Content generation approaches are commonly categorised as constructive, generate-and-test or search-based. Constructive approaches work in a single pass and generate content in a predictable and typically short time. Several classic PCG algorithms are constructive in nature, e.g. Perlin noise, L-systems and variations on dungeon diggers, and constructive PCG is widely used in commercial games for "decorative" elements such as skyboxes and plants [16]. However, when generating content with playability constraints (also called "necessary" content), such as puzzles that need to have a solution, maps that need to be balanced or levels that need to be winnable, constructive approaches run into the problem that there is typically no way to guarantee such properties. To the extent that constructive methods are used for such content, the expressive range of the algorithms tend to be severely curtailed in order to avoid the generation of unplayable content; see for example the unimaginative generated levels in many roguelikes and infinite runners.

One solution to this problem with constructive algorithms is to generate-and-test: apply some sort of test (is the map balanced? the puzzle solvable?) and keep regenerating until the content is good enough. However, depending on how strict the test is, it might take very long time until acceptable content is generated.

A more informed version of generate-and-test is search-based PCG. Here, an evolutionary algorithm or other stochastic global search algorithm is used to search content space for content that optimally satisfies some evaluation function – for example, map balance or level winnability. Evolutionary algorithms have excellent facilities for generating sets of diverse content artefacts, even while satisfying multiple fitness functions [17]. However, search-based approaches are still in general much slower than constructive approaches, often too slow to be used in real time. To make matters worse, the more sophisticated the playability demands are, the more computationally expensive the evaluation function becomes. In particular simulation-based evaluation functions which require an agent playing through part of the game are very time-demanding.

In this study we present a new attempt at combining the diversity, flexibility and playability-preserving ability of search-based approaches with the speed of constructive approaches. We call this a *progressive* approach to the generation of playable content. The first step is to turn the CG problem on its head: instead of generating playable content directly, we first generate a timeline of in-game interactions that need to be performed in order to successfully play through the content artefact. This is done using a search-based approach, but as we can use a simple evaluation function based on lightweight simulation this part can be done quickly. Every time a timeline is evaluated, it is turned into level content. This is done using a constructive approach, where each point in the timeline is converted into a part of the level. As we shall see, this allows fast and reliable level generation with a considerable expressive range.

This study (in Chapter 5) presents the components of this approach in more detail. It then presents a case study of the application of the progressive method to generating puzzles for the physics-based puzzle game *Cut the Rope*. Finally, we report some observations on the performance and expressive range of this method in the given domain. This is discussed in Chapter 5.

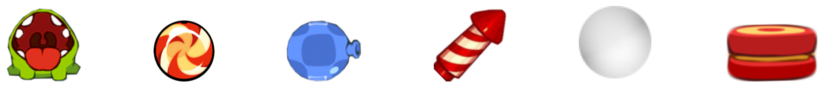
— 3 —

Testbed Game: Cut the Rope: Play Forever

3.1 Cut the Rope

In this study, *Cut the Rope* (CTR), will server as the main testbed game for our novel methods to test their applicability and performance. CTR is a very popular commercial physics-based puzzle video game. It was released in 2010 by ZeptoLab for mobile devices. The game was a huge success and it has been downloaded more than 100 million times.

The gameplay in CTR revolves around feeding candy to a little green frog-like monster named *Om Nom*. The candy is usually attached to one or more ropes which have to be cut with a swipe of the finger to set it free. Auxiliary objectives include collecting as many of the stars present in the level as possible. All game objects obey (a digital approximation of) Newtonian physics; in particular, they are affected by gravity [18]. The player loses the game by letting the candy escape (e.g. fall) outside the level boundaries. Each level can be seen as a puzzle, as guiding the candy to Om Nom is complicated by the presence of obstacles and other physics-based elements that block or redirect the candy. The set of components included in the original game includes air-cushions, bubbles, rockets, spikes, spiders, suction cups among others (see Fig. 3.1.) The player interacts with the game by cutting a rope, tapping an air-cushion, a bubble or a button triggering a sequence of physics-based consequences.

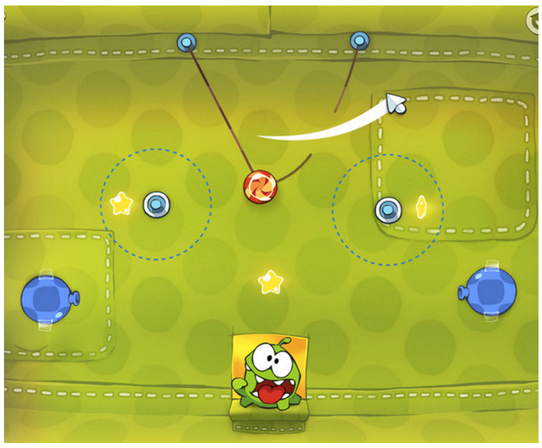


(a) *Om Nom* **(b)** *Candy* **(c)** *Air-cushion* **(d)** *Rocket* **(e)** *Bubble* **(f)** *Bumper*

Figure 3.1 – *The various components presented in the original Cut the Rope game and considered in our clone.*

3.2 CRUST 2D Physics Engine

There is no open source code available for the game so we had implemented our own clone using our CRUST engine [19] and the original game art assets. The physics engine used for the design and generation of our game is implemented in C# with XNA


(a) *A screenshot from the original game*

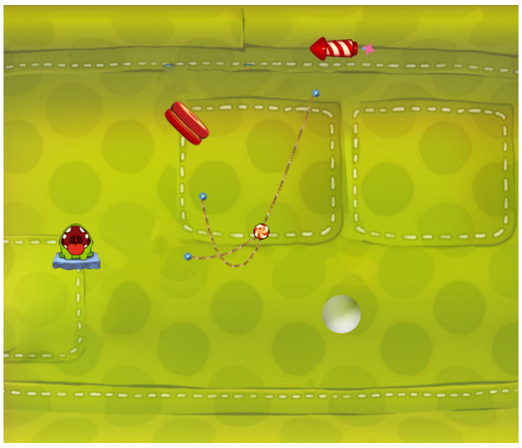

(b) *A screenshot from our clone of the game*

Figure 3.2 – *Two snapshots from the original Cut The Rope game (a) and our clone version (b) showing Om Nom waiting for the candy which is attached to ropes.*

for managing runtime environment. The engine is a modified version of the Millington Engine [18]. Our modifications include adapting the engine to work with 2D environment and implementing the spring constraint. In its current state, the engine is able to provide efficient handling for physics simulations. The engine implements impulse force collision modelling to deal with rigid objects. Other physics-based motions such as springs, ropes, and hard constraints can also be simulated in the current version. The engine is also facilitated with a friendly user interface that allows editing objects and their physical properties at run time. Our clone, *Cut The Rope: Play Forever*, does not implement all features of the original game, but includes all the fundamental features and allows faithful reimplementation of a large portion of the original levels. Fig. 3.2a shows one of the level in the original game while Fig. 3.2b presents a level from our clone Cut The Rope: Play Forever.

— 4 —

A Projection-Based Approach for Real-time Assessment and Playability Check for Physics-Based Games

4.1 Methodology

In this chapter, we introduce an authoring tool for physics-based puzzle games that supports game designers through providing visual feedback about the space of interactions. The underlying algorithm accounts for the type and physical properties of the different game components. An area of influence, which identifies the possible space of interaction, is identified for each component. The influence areas of all components in a given design are then merged considering the components' type and the context information. The tool can be used offline where complete designs are analyzed and the final interactive space is projected, and online where edits in the interactive space are projected on the canvas in realtime permitting continuous assistance for game designers and providing informative feedback about playability.

4.2 The Influence Area

The fundamental concept in the method proposed here is the *Influence Area* (IA) Each component in the game has its own physical properties that affect the candy's trajectory, velocity and acceleration. The effect takes place when the candy falls within the IA of that component. A component's IA is is all possible trajectories the candy can follow when interacting with this component. An estimation of the IA is defined differently for each component according to its type as follows:

- Rope: The IA covers the candy's possible trajectories when detached from the rope. Since the physical properties of a rope resemble those of a spring, the rope's IA is defined as a trapezoid where its two legs are defined by the two lines passing through the rope's pin and one of its bases is the horizontal line passing through the candy (see Fig 4.1a for an illustration).
- Air-cushion: Its IA is a trapezoid oriented in its blowing direction: one of its bases is the vertical line passing through the air-cushion, the distance to the other bases

is defined based on the air blowing force; one of its legs is defined by a line passing through the air-cushion and angled slightly downwards due to the gravity force while the IA extends to the boundary of the canvas. An illustration is presented in Fig. 4.1b.

- Bubble: The basic IA of a bubble is the simplest and is defined as a rectangle surrounding the bubble and extending upwards to the canvas boundaries. This covers the area of the bubble while floating and in the case of being pressed freeing the candy downwards as presented in Fig. 4.1c
- Rocket: Based on the rocket's direction, its IA is a trapezoid covering the candy's possible paths when set free. Consequently, the bases of the trapezoid are the vertical line passing through the rocket and the boundary of the canvas, its legs are the other canvas' boundary and a line starting at the rocket's position and angled according to its direction as can be seen in Fig. 4.1d
- Bumper: The IA of the bumper is the hardest to define due to its complex physical property and the dependency between its IA and the direction of the collision with the candy. The basic IA is defined as a six-point polygon covering all possible bouncing trajectories after the collision considering the effect of gravity. An example can be seen in Fig. 4.1e.

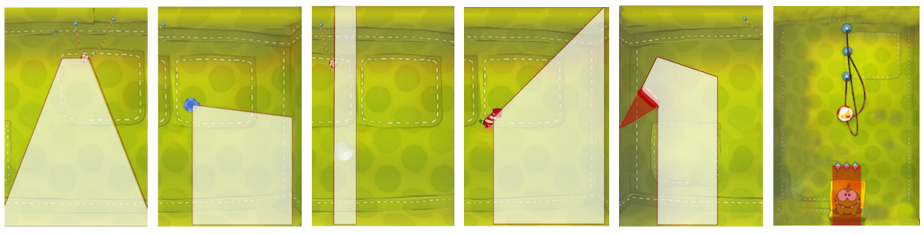

(a) (b) (c) (d) (e) (f)

Figure 4.1 – *Examples of the basic influence areas for the different game components. Inclusion areas are presented in light colour while exclusion areas are in red.*

The interactive space in a given design can then be identifying by combining the IAs of all components presented. We call this the *Projection Area* (PA). Note that the previously defined areas are carefully chosen approximations of the exact areas which could be more accurately determined by applying the laws of physics. We decided to consider these approximations instead of the exact calculations since they yield almost equally accurate results while being considerable computationally cheaper.

While the previously defined IAs are relatively easy to calculate, these are the most basic form of IAs. For complicated designs with more than one component, the IA of each component will typically be altered by the presence of another. Things get more complicated as the number of components increases and a more advanced solution becomes a necessity. The solution we propose differentiates between two types effects according to how the components' IAs interact: components of *inclusive* and *exclusive* effects. Components with inclusive effects are characterised by IAs that increases the PA already established either by expanding or by adding a new separated range. Exclusive effects, on the other hand, decrease the size of the PA through subtracting parts that became inaccessible as a result of adding a component with such property. Some of the components exhibit properties from the two types depending on their type and the other component of interaction. This applies for example to bumpers which expand the area in the direction that faces the candy and exclude the one in the other direction. Fig. 4.1 presents some examples for both cases.

Given the components' characteristics and IAs, an agent can be implemented to provide an online visual feedback and playability check of complete designs or while the level is being designed through automatically expanding and/or reducing the final playable area as new components are being added or removed.

In the following sections, we describe the steps followed to identify the projection area using a projection agent.

4.3 The Projection Algorithm

The heart of the proposed method is the projection algorithm. This algorithm handles the interactions between the components and effectively and accurately defines the interactive area. The basic idea is that by starting with the ropes to which the candy is attached, and recursively expanding and/or subtracting fragments of the PA, we can ultimately deduce the area that is reachable by the candy. Eventually, we can assess whether a level is playable by examining the overlap between the resulting PA and Om Nom's position. If such an overlap exists, the level is playable. We considered a number of implementation choices when constructing the agent. In order to support these choices and clarify the reasoning behind them, we will start by explaining two scenarios where a naive implementation of the basic idea will not work.

Scenario 1: Order effect

Combining the IAs of all components in one pass following Equation 4.1 to build the final PA is not a feasible solution. This is mainly due to the nature of the game where the IA of a component depends primarily on the candy's current position, direction, and velocity and the order in which components are activated.

$$PA = \bigcup_{i=1}^{Components} IA_i \tag{4.1}$$

Consider an interaction between two ropes, two bumpers, two rockets and the candy as an example (Fig. 4.2). Applying the IAs of the components in a different order will lead to different results. For instance, if we apply the IAs in the order: ropes, rockets and finally bumpers, the result will be the PA presented in Fig. 4.2a. As can be seen in this case, the IAs of the bumpers block the candy from reaching one of the rockets that lead to Om Nom and as a result, the level will be classified as unplayable. On the other hand, if we follow another order so that the IAs of the rockets are applied last, the resultant area will cover Om Nom and the level will be playable (Fig. 4.2b).

The order in which the components are handled is therefore essential since different order will lead to different PAs and consequently different judgement about playability.

While it is vitally important to determine the sequence in which the components should be executed, this order is not obvious.

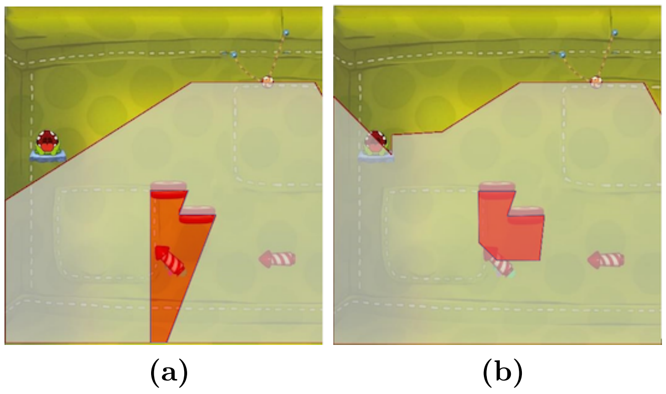


Figure 4.2 – *Two different results of applying the IAs in different orders. The first image shows the IAs when applying: ropes, rockets and finally bumpers in order. The second image shows the IAs when applying: ropes, bumpers then rockets.*

Scenario 2: Solution path

One possible solution to the problem described in the previous section is to process the components according to the possible paths that could be followed to solve the game. This approach will also fail since it is highly likely that there will be more than one component that could be activated at any specific time during the game. The right order in which these components are handled can not be easily accurately defined (we will end up facing the same issue discussed in the first scenario). Furthermore, our goal is to define an area that constitutes all reachable positions and not only the ones that lead to the solution and this can not be achieved by relying solely on the solution path.

4.3.1 The Projection Algorithm

The above mentioned scenarios emphasise the need to a more intelligent strategy that considers rich context information and handles the ordering effect. This can be achieved by constructing an agent that processes the components recursively one at time, analysing the game state and considering the possible interactions with other components in a depth-first approach until a stopping criteria is satisfied. The algorithm proposed is a search method that exhaustively traverses the full tree of possible paths. The resultant PA is the combination of all PAs for all paths explored. This permits extraction of the the full playable area (those interaction areas that lead to a solution and those that do not). We intentionally do not discard unsolvable paths since our goal is to help designers visualise the area that is used and in which players can take actions regardless of their outcome. The algorithm followed (Fig. 2) can be described as follows:

The algorithm handles two main data structures: a sorted list of components, *coveredComps*, and a PA (polygons represented as a list of points) that is constantly updated as new paths are explored. The algorithm starts by calculating the IAs of the ropes since

Algorithm 1: The Projection Algorithm

```
  Data: List of components with their initial positions;
  PA = ropes.applyIA();
  coveredComps = PA.findCovered(allComps);
1 while (not(empty(coveredComps)) do
2     coveredComps = PA.findCovered(allComps);
3     coveredComps = removeExcluded(coveredComps);
4     for cc in coveredComps do
5         PA += cc.applyIA();
6         projectionAlgorithm(PA);
```

the game always begins by cutting them to initiate the candy's movement. The components covered by the ropes are calculated and added to a set of covered components. The algorithm then iterates until the tree representing all possible paths is explored. In each iteration, a set containing the components covered by the IA of the currently processed component is calculated (line 2). The algorithm then checks if any of the covered components can be excluded (line 3) and removed from the set. A recursive call is then performed; the IA for each item in the set is calculated (line 5) followed by a recursive call to the algorithm with the newly formed PA (line 6).

Note that although the algorithm performs exhaustive search, the method takes on average 0.1 ± 0.02 sec (over 100 runs) to process complete designs of an average of 8.53 ± 1.74 components. This is because only few components become cover in every iterations and each level usually contains a small number of components. Fig. 4.3 presents an example level and the full tree explored to calculate the PA. A step-by-step example of how the PA is calculated is presented in Section 4.7.1, however this example is presented to draw attention to the size of the tree and to clarify the very short amount of processing time required to traverse it.

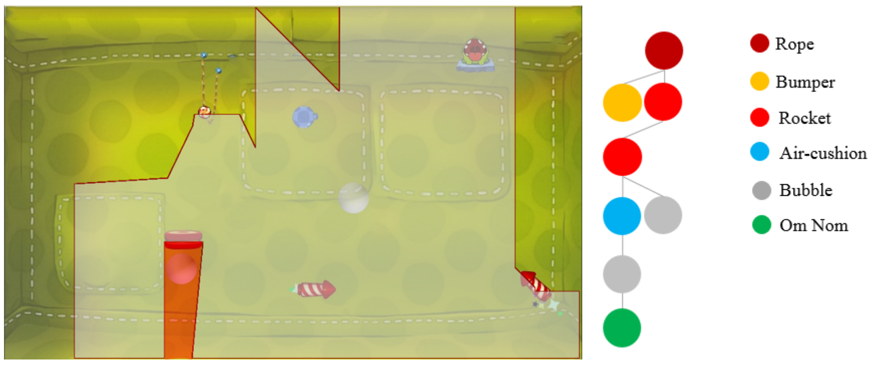


Figure 4.3 – *Example level and the corresponding explored tree.*

The success of the projection algorithm depends on the following important details:

Estimating the Interaction Direction

An important factor that defines the IA of a component is the direction of interaction with the candy. For example, the IA of a bumper is reversible according to the direction of the collision. Simulating the game is one possible way to precisely set the direction. However this solution is too slow in many cases (an average of 29.87±58.28 sec is required to simulate one level [9] and the simulation has to be repeated for every edit). Moreover, we are not interested in the exact position but with a reliable estimation of the direction. For these reasons, the candy's direction is calculated according to the position of the last component with which the candy interacted. The algorithm proposed solves this issue by considering the order in which they become covered. Specifically, the direction of interaction with a component n is the position of the component $n-1$ in the recursive call.

Handling Context

We previously discussed the basic IA of each component and mentioned that the IA of one component is usually affected by the context. In the following, the IAs of some components are refined to account for their interaction (the IAs of the other components remain intact):

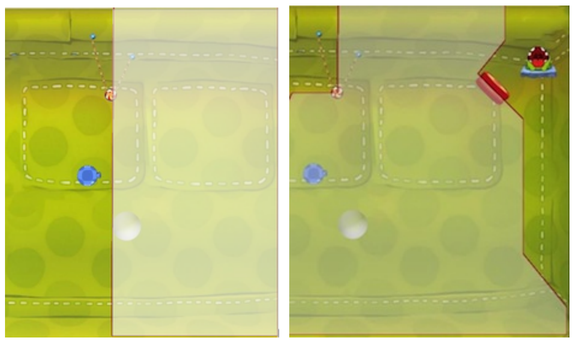
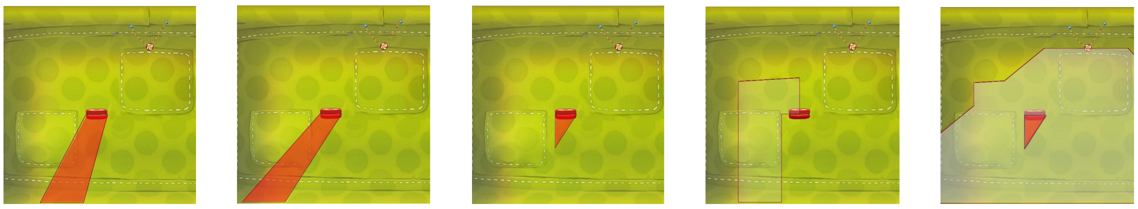


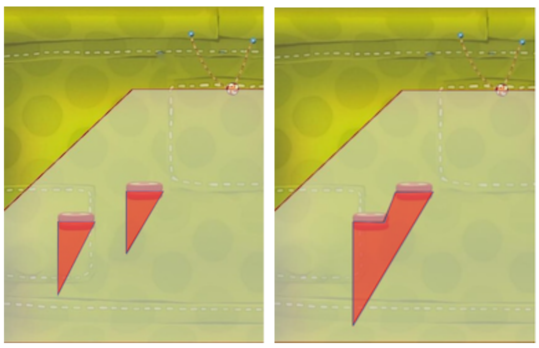
Figure 4.4 – *The extended influence area of a bubble being covered by an air-cushion (a), and a bumper being covered by a bubble (b).*

- Bubble: The presence of other components such as an air-cushion, a bumper or a rocket changes the IA of a bubble. When covered by a an air-cushion for example, the IA of the bubble extends so that the new area is a wider rectangle due to the effect of the air force. This is seen if Fig. 4.4a.
- Bumper: The IA of a bumper is altered relative to the type of the other components placed nearby. Since bumpers have expansion and exclusion areas, they are the hardest to determine. A bumper exclusion area should be re-calculated in respect to all components that can reach it. That means bumpers change their expansion and exclusion areas every time a component can reach them (i.e. the expansion and exclusion areas are re-calculated in respect to all components reaching the bumpers.) An example is shown in Fig 4.5 where multiple processing stages are



(a) *Initial exclusion area generated from the first rope.* **(b)** *Initial exclusion area generated from the second rope.* **(c)** *The initial exclusion area being cut by candy's gravity trajectory area.* **(d)** *The expansion area of the bumper.* **(e)** *The final PA of the whole design.*

Figure 4.5 – *The multiple stages of obtaining a bumper's final Influence Area.*



(a) **(b)**

Figure 4.6 – *Examples of the IA of two bumpers according to the distance between them. In (a), the bumpers are processed separately while in (b) their exclusion areas are combined since they are closer.*

needed to determine the final IA of a bumper. Also the existence of a nearby bubble and an air-cushion, for instance, results in a new extended area as can be seen in Fig. 4.4b. Also, the presence of another nearby bumper forms a new combined exclusive area as can be seen in Fig. 4.6.

When accounting for the above factors, the projection algorithm is capable of effectively constructing the PA and successfully detecting when the design is playable.

4.4 Implementation Details

The Clipper library [20] is used to preform polygon clipping. We used the C# version of the library and interface it with our physics engine (CRUST) to simulate the game.

4.5 Ropossum Integration

The projection algorithm is incorporated as part of the authoring tool *Ropossum* [8]. Some of the functionalities previously provided are the automatic generation of playable content [15] and the automatic check for playability [9]. With projection areas as a newly added feature, designers can benefit from editing procedurally generated levels and visualising PAs of complete designs or during the level generation process[1]. The method also supports playability check which can be done faster than previous methods (see section 4.10.)

4.6 Online and Offline Assessment

The projection algorithm presented and the scenarios discussed assume that the components and their positions are known and provided as input to the algorithm. In this case, the algorithm is used as a tool for post-design assessment of a level where the final interaction area is projected to give the designer a visual feedback. This information can be used to alter the initial design and changes to the PA can be visualised accordingly either in the offline mode (after all adjustments are made) or through the online mode.

In the online mode, the projection algorithm is activated while the scene is being edited. In this case, whenever a new component is placed in the canvas, the algorithm is called with the list of all components used so far. If the newly added component has an exclusive property, the PA is recalculated through a new run of the algorithm. Otherwise, the PA is expanded according to the type of the recently added component. This might result in covering other components in the scene and therefore the algorithm is called with the set of covered components only as input.

The IAs of the components were carefully tuned and the method has been tested on several cases and showed promising results in terms of accurately estimating the PA and identifying solvable designs. There is however no guarantee that there are no levels on which the algorithm fails ((un)fortunately, we cannot report any). If such situations exist, we do not know whether the algorithm is biased towards false positives (cases where the method classifies unsolvable levels as solvable) or false negatives.

4.7 Showcases

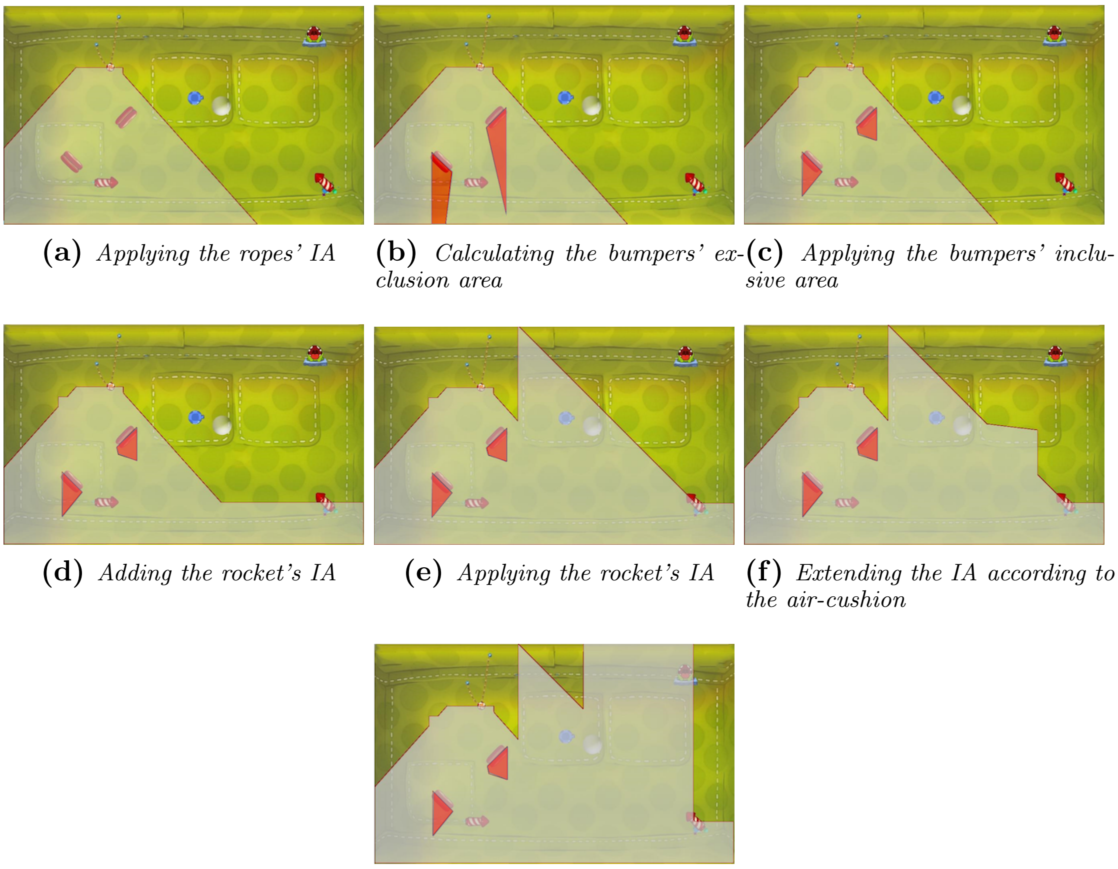
Below we present two examples that showcase the algorithm's capabilities during both the offline and the online mode. Note that the algorithm works exactly the same in both cases and we differentiate between them to clarify and emphasise its capabilities.

4.7.1 Showcase 1: Offline Mode

In this example, a complete design is provided as input to the system to calculate and visualise the resulting playable area. Fig. 4.7 presents an illustration of the steps followed by the algorithm. We start by calculating the ropes' IAs which in this case cover two bumpers and a rocket (Fig. 4.7a). Since the algorithm sorts the components so that bumpers are handled first, the areas excluded by the bumbers according to the candy's current position are then subtracted from the PA (Fig. 4.7b). Their coverage areas are then added to the PA, reincluding fragments of the area previously excluded (Fig. 4.7c).

[1]A video showing the implementation of the framework in CTR is available online: http://www.mohammadshaker.com/ropossum.html

Continuing to handle the rest of the component, the IA of the covered rocket is then applied extending the PA to cover the other rocket (Fig.4.7d). The IA of the newly added rocket covers the air-cushion and the bubble (Fig. 4.7e). Recursively handling these two components results in first adding the IA of the air-cushion (Fig. 4.7f) then expanding the resultant area according to the IA of the bubble (Fig. 4.7g). At this stage, and since there are no more components to handle, the process is terminated and the level is considered playable. The visualisation of the PA in this case clearly illustrates that all components are accessible starting from the candy's initial position. It also reveals the active space of the canvas where actual interaction can take place.



(a) *Applying the ropes' IA*

(b) *Calculating the bumpers' exclusion area*

(c) *Applying the bumpers' inclusive area*

(d) *Adding the rocket's IA*

(e) *Applying the rocket's IA*

(f) *Extending the IA according to the air-cushion*

(g) *Adding the IA of the bubble*

Figure 4.7 – *The offline use of the algorithm. Projection area is presented in a light colour and excluded areas are presented in red.*

4.7.2 Showcase2: Online Mode

The online use of the algorithm is illustrated in Fig. 4.8. We start by a simple level design of one rope and Om Nom (Fig. 4.8a). The player/designer can add, remove and/or reposition components using an authoring tool [8]. A green flag (top left) is used to indicate a playable design when adding a second rope (as in Fig. 4.8b). During the design process, the PA is continuously updated as the game state changes. Fig. 4.8c, 4.8d and 4.8e presents the updated PA after positioning three bumpers leading to the exclusion of Om Nom and as a consequence a detection of an unplayable level. This situation is overcame when the designer adds an air-cushion (Fig. 4.8f) which resets the flag.

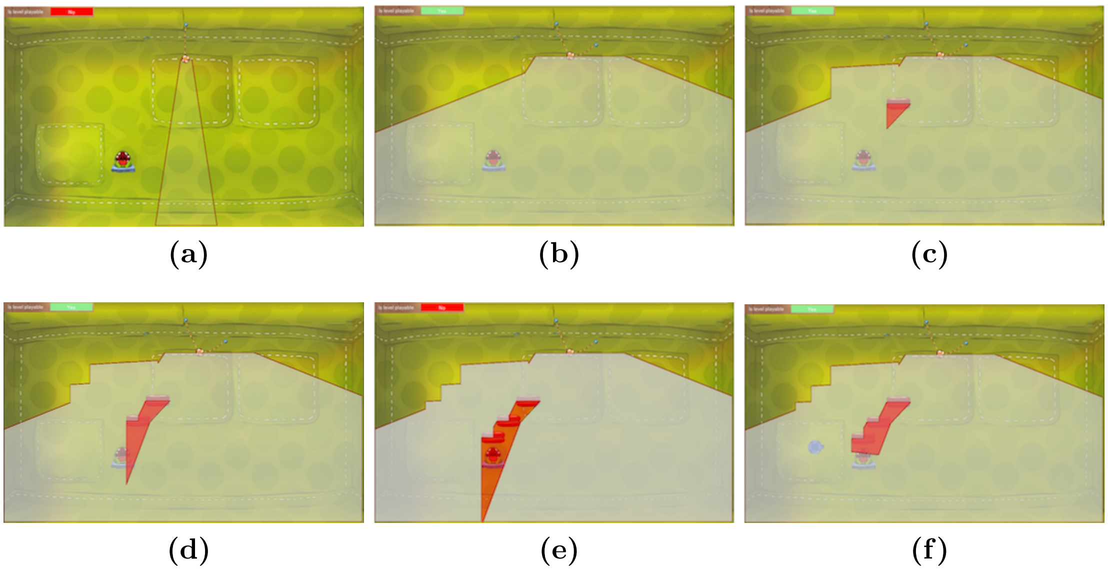


Figure 4.8 – *The online use of the algorithm in Ropossum authoring tool. Projection area is presented in a light colour and excluded areas are presented in red. The flag on the top left of the screen indicates, in real-time, the playability score of the current design; Red for unplayable designs and Green for playable ones. This is changed while the designer is prototyping in real-time.*

4.8 Evolving Playable Content with The Projection Agent

We have exploited so far how the projection agent can be used to visualise the playable area of a given design and check for its playability. Since the agent can be used for playability check, we can instantly think further to use it as a building block for a new approach for the generation of content for physics-based games.

The new approach builds on the previous work by [15] and [9] on procedurally generating content for the game using a search-based approach. In [9] a simulation-based, Prolog agent was used to play through the level assigning its playability score. Although the agent was able to accurately detect a playable design, its performance overhead capitalise over its usability for real-time systems (checking a single level takes an average of 29.87 ± 58.28 sec). The simulation-based agent was used to evaluate level structures proposed by a Grammatical Evolution software. Since the main problem with this is the performance overhead, and since our new approach excels in this area, we can simply replace the simulation-based agent by the projection-based agent to assign the level's playability score, improving the performance of the model.

Building on the framework discussed in [9], and by replacing the simulation-based agent by the projection-based agent, we can derive a new approach for content generation for physics-based games. The projection agent will check a level playability and assign

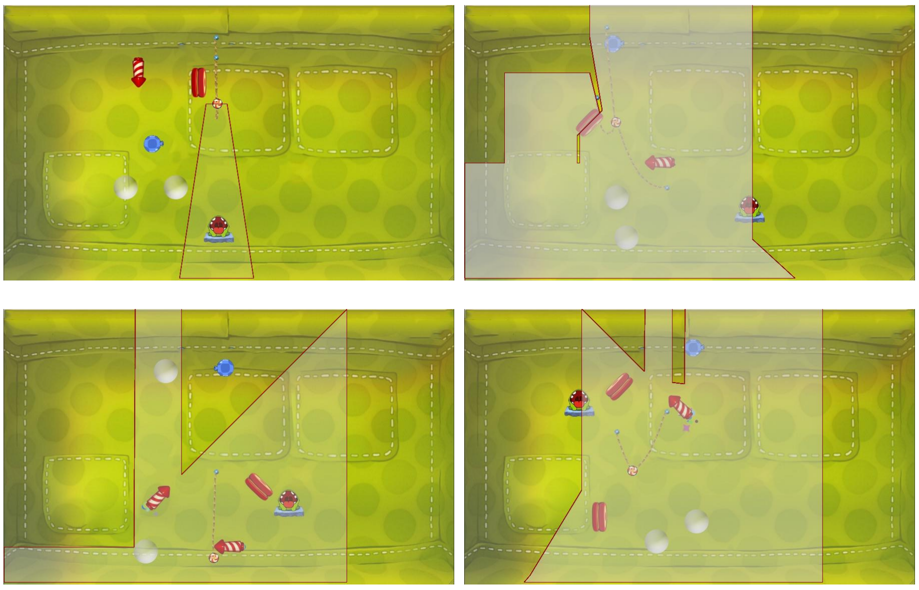


Figure 4.9 – *Examples of generated levels of increased difficulty*

its fitness accordingly. If the level is playable, it is rewarded a perfect score, if not; the shortest distance between Om Nom and the PA will be considered as a penalty. Fig. 4.9 shows a different generated levels samples of different difficulties.

4.8.1 Experimental Setup for Evolving Playable Levels

The GE experimental parameters used for evolving levels are the following: 100 run of 100 generations with a population size of 20 individuals. The ramped half-and-half initialization method was used and the maximum derivation tree depth was set at 100, tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.7, and 3 maximum wraps were allowed. The evolution process is terminated when a playable level is found or when the maximum number of generation is reached.

4.9 Results and Analysis

In order to test the agent, a series of analysis has been conducted to thoroughly test the different characteristics of the projection-based. Some analysis has been conducted in a previous work [15].

4.9.1 Projection Area Map

To facilitate a more in-depth insight on the differences between the generated levels we converted all of the 100 levels' Projection Areas into one color map. This image is generated by assigning a value for each pixel which is the average color value of all pixels in the same position in the full set of levels. Fig. 4.10 shows the projection area map across 100 generated levels. It shows that the projection area of most levels are condensed at the centre. This is mainly because of ropes' IAs covering wide range of area when there are more than one in the level, particularly at the center.

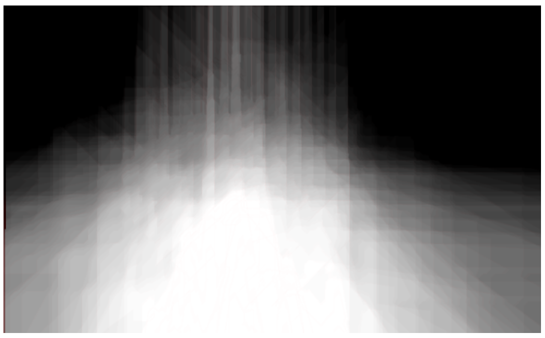


Figure 4.10 – *The projection area map of 100 generated levels*

4.9.2 Axiality

Axiality relates to the orientation of the items in a level. A level with maximum axiality in an axis has all components oriented in parallel to that direction. The feature values are normalised to the range [0,1] using min-max normalisation. The component distribution on both axes according to this measure gives an idea of how easy the level is [15]. For example, a level with high axiality scores on both axes is usually harder to play and requires more thinking since this configuration means that to solve the level, the player should make use of the different components and the candy should travel a long distance before it reaches Om Nom.

The axiality of a level is measured by projecting the components on the x and y axes and measuring the distance covered. The axiality is then represented as a point in a two-dimensional space where the axes represent the distances. Fig. 4.11 illustrates the distribution of all the levels generated according to the axiality measure. The color of each square indicates the number of levels generated with the corresponding distance covered on both axes. A level with a low score on one of the axis points out to a small range of coverage on that axis.

The figure illustrates a clear bias in the axiality measure towards generating levels of high axiality on both axes. This indicates that in most of the 100 levels generated, the components are placed within a large distance on both axes. This is most likely a result of the design of fitness function which is biased against levels that do not preserve the minimum distance allowed between components producing levels with components scattered around.

4.9.3 Density

A level has a high density if the components presented are placed within a very close distance to each other or when the components are gathered in high compactness groups. The more the components and the closer they are to each other, the higher the density of the level [15]. This measure takes the same 3 x 3 regions as the diversity measure to calculate the level's density. The density is calculated as the standard deviation of the regions that contain at least one component according to the equation:

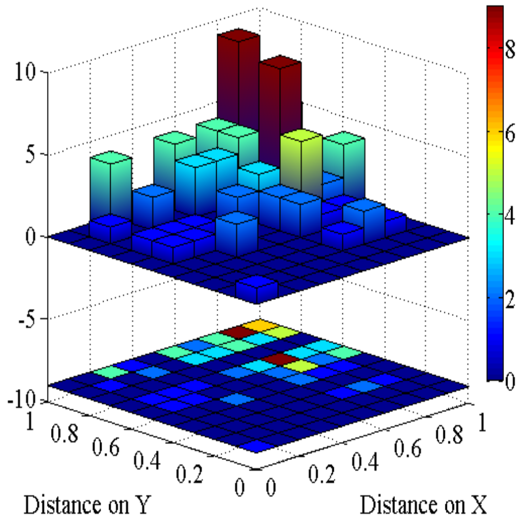


Figure 4.11 – *The distribution of all 100 levels generated according to the axiality measure. The x and y axes represent the distance the components cover on the corresponding axis in the level. The color in each square corresponds to the number of levels generated that has the associated distance cove*

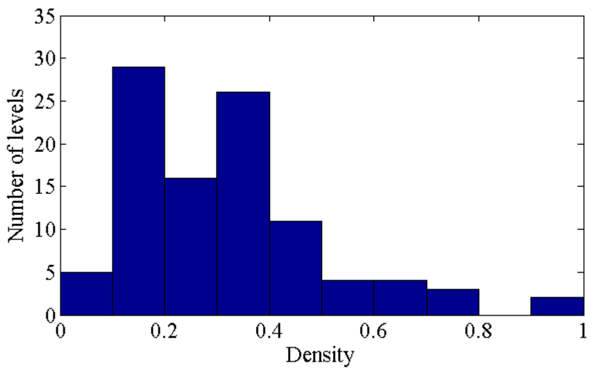


Figure 4.12 – *The histogram of the density measure for the 100 levels generated*

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

where n is the number of non-empty regions, $\bar{x}$ is the average value of components placed in the n regions and x_i is the number of component in the region i [15]. The distribution of levels according to the density measure is presented in Fig. 4.12 where the feature values are normalized to the range [0,1] using min-max normalization According to Fig 4.12, a clear bias is observed towards generating levels of relatively low density scores. The results can be explained by the the projection algorithm favouring levels with scattered components with better defined influence areas.

4.9.4 Axiality vs Density

To investigate the expressive range of the generator along more than one measure, we generated a graph that shows the distribution of levels along the axiality and density dimensions [15] as can be seen in Fig. 4.13. The axiality score in this experiment is calculated as the Euclidean distance between the x and y distances calculated in Section 4.9.2.

The figure shows that the high majority of levels have low density and relatively high axiality score in both agents. It is worth noticing that for very low density values, levels of average to high axiality score can be generated. This is expected since a level of low density indicates that the components are scattered around the level and therefore they will most likely cover a wide range on both axes resulting in a high axiality score.

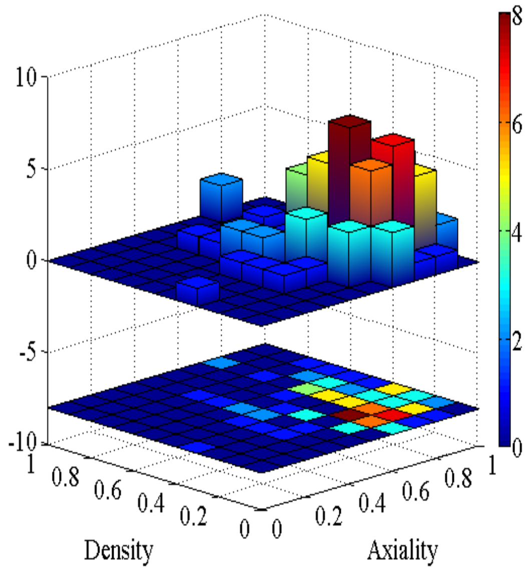


Figure 4.13 – *The number of levels generated according to the axiality and density measures.*

4.10 Performance Comparison

To better demonstrate the differences in performance between the agents found in literature and our new approach, we assess all of them by the time it takes the agent to check a level for playability. Table 4.1 shows the performances across different agents:

1. Pseudo-Random [15]: This agent use a set of random actions in order to solve a level.
2. Simulation-Based [9]: Using first-order logic (Prolog) and a simulation-based approach, the agent can infer what action should be activated in each time step to solve the level.
3. Projection-Based: Using components characteristics, physical properties and their relative positions, the agent can build up the IA of each component and infer the level's PA, therefore, the playability of the level.

Table 4.1 – Comparison between the current approach and the previous ones. The table presents the time required to check if a design is playable (in seconds), the time required to evolve a playable design (in seconds) and the number of generations required for a valid design.

Agent	*Playability*	$Evolution_{time}$	*Generations*
Pseudo-Random [15]	0.98 ± 0.64	21.27 ± 23.44	2.63 ± 1.82
Simulation-Based [9]	29.87 ± 58.28	470.1 ± 525.4	2.48 ± 1.58
Projection-Based	0.1 ± 0.02	13.1 ± 7.57	1 ± 0

Table. 4.1 clearly shows how the projection agent is faster than the simulation-based agent for playability check and for evolving playable content. Though, a future direction should better measure the projection agent's accuracy against the simulation-based agent and the quality, the difficulty and the fun factors of the generated content produced by both.

4.11 Applicability of the Projection-Based Approach

Driven by our motivation to apply the projection-based approach on other games, one could think of countless physics-based game where this approach can be a vital asset for designers. One of which is the most successful physics-based mobile game of all time; Angry Birds. Fig. 4.14 shows and illustration of the first outcome after the first execution loop of the projection algorithm in Angry Birds. The PA shown in the figure can be extended further into inclusion and exclusion areas in the same manner discussed in this study, taking into account that blocks in angry birds are similar to bumpers in CTR. Both can exclude some other components and both can interact with the game's main character (the candy in CTR, and the bird in Angry Birds) through bumping and bouncing.

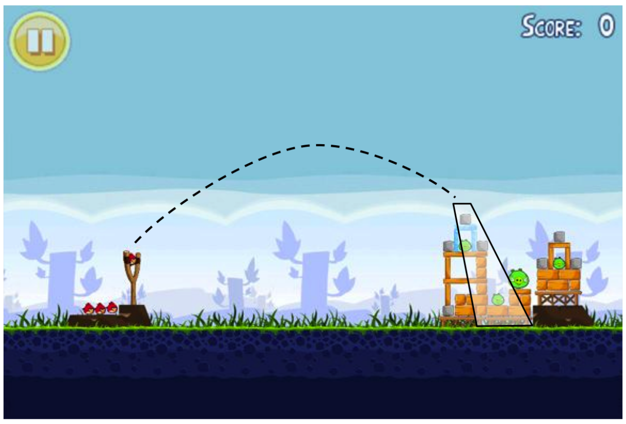
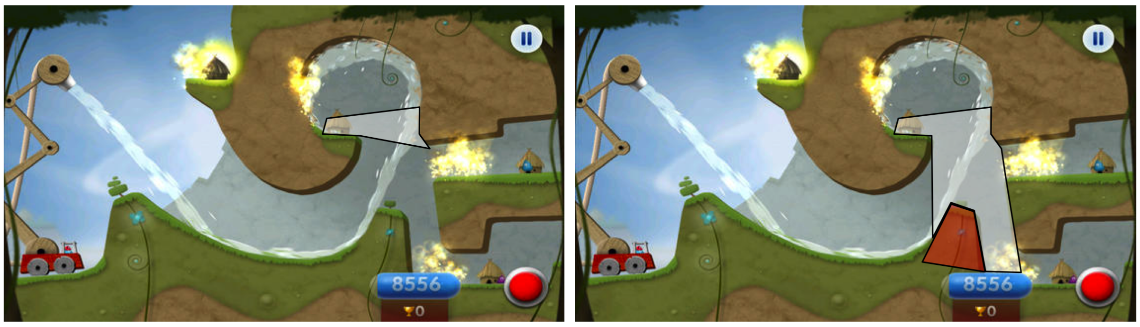


Figure 4.14 – *A showcase of the first phase to obtain the bird's IA in Angry Birds.*

Fig.4.15 shows another example of an IA of a structure in another famous physics-based mobile game; Sprinkle. The main idea of Sprinkle is to lead the water from the bumper to reach a different area of the level by controlling the angle of which the water



(a) **(b)**

Figure 4.15 – *A showcase of the first and second phase to obtain a structure's IA in Sprinkle.*

is bumped. This resembles a harder task for the projection approach. Though once a level structure is given, it can be decomposed into different *structures*. Each structure can have its own inclusion (fig.4.15a) and exclusion (fig. 4.15b) areas. The sum of all these can lead to the final PA of the level by following the projection approach discussed in this chapter.

The figures fuel our earlier motivation to design an algorithm that's able to handle different game styles in the physics-based games genre. Once the inclusion and exclusion areas of each component is known in a game, the approach can be applied to handle the playability check of the design. Applying the approach for different games is an interesting future direction that can show the approach's applicability, strengths and weaknesses.

— 5 —
A Progressive Approach to Content Generation

5.1 Methodology

Procedural Content Generation (PCG) approaches are commonly categorised as constructive, generate-and-test or search-based. Each of these approaches has its distinctive advantages and drawbacks. In this chapter, we propose an approach to Content Generation– in particular level generation – that combines the advantages of constructive and search-based approaches thus providing a fast, flexible and reliable way of generating diverse content of high quality. In our framework, CG is seen from a new perspective which differentiates between two main aspects of the gameplay experience, namely the order of the in-game interactions and the associated level design. The framework first generates timelines following the search-based paradigm. Timelines are game-independent and they reflect the rhythmic feel of the levels. A *progressive*, constructive-based approach is then implemented to evaluate timelines by mapping them into level designs. The framework is applied for the generation of puzzles for the *Cut the Rope* game and the results in terms of performance, expressivity and controllability are characterised and discussed.

5.2 Evolving Game Content: A Progressive Approach

The framework we propose (presented in Fig. 5.1) consists of two layers: a Timeline Generator (TLG) and a Game Simulator (GS) where the generated timelines are evaluated and scored according to playability and/or other design constraints. The framework is search-based in the sense that the structures of the timelines are evolved by the TLG. Each individual is then simulated by the GS following a constructive approach (using a game specific software) and assigned a fitness according to predefined criteria. Evolution then continues to explore the generation of "better" timelines.

5.2.1 Timeline Generation

We define a level timeline as a sequence of in-game interaction events that should occur at specific times throughout the game session. By taking all those actions at the specified times the level can be played through successfully (there might or might not be other equally successful timelines for a given level). For real-time games, the timeline is essential for meaningful gameplay experience as it reflects the rhythm of playing a level. The

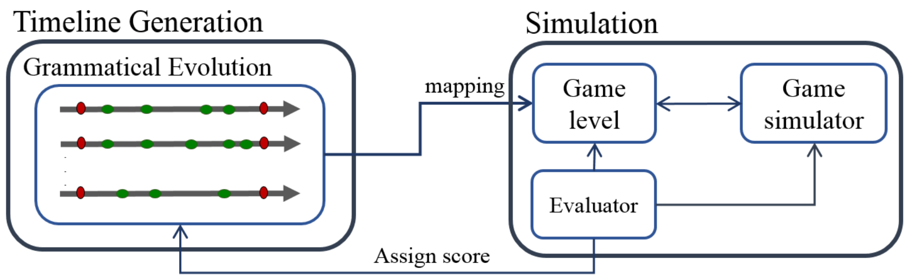


Figure 5.1 – *The progressive content generation framework. Timelines are evolved by grammatical evolution and evaluated by the game simulator. The simulator progressively maps a timeline to a game design through simulating the game. A complete design is finally scored based on the result of the simulation and other design aspects.*

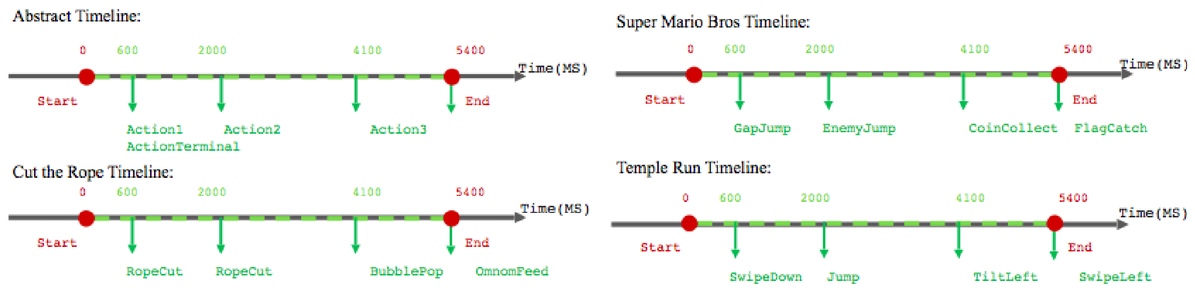

Abstract Timeline:
0 600 2000 4100 5400
Time(MS)
Start End
Action1 ActionTerminal Action2 Action3
Super Mario Bros Timeline:
GapJump EnemyJump CoinCollect FlagCatch
Cut the Rope Timeline:
RopeCut RopeCut BubblePop OmnomFeed
Temple Run Timeline:
SwipeDown Jump TiltLeft SwipeLeft

Figure 5.2 – *An abstract timeline and its possible interpretations in different games.*

timelines in our framework are defined in a generic way that can be easily interpreted and applied in dissimilar games. Fig. 5.2 shows different example timelines for different games. The first timeline contains abstract actions that can be instantiated into game-specific actions such as the instances presented for *Super Mario Bros* (SMB), *Cut the Rope* and *Temple Run*.

The events in a timeline are presented given their temporal order (activation order.) Some games are more order-sensitive than others. For instance, in games where the player can navigate in both directions, such as SMB, the order of placing the components is important from a design and aesthetics point of view, while in other games, such as CTR, the order in which the components are activated plays a key role in making the game playable (and in how difficult it is).

We use grammatical evolution [21] to define and evolve timelines since it provides a simple way of defining phenotype structure through the use of the design grammar. Grammatical evolution also facilitates adapting the framework to other games, as this requires only instantiating a new design grammar with the game-specific events.

Grammatical Evolution

Grammatical Evolution (GE) is the result of combining an evolutionary algorithm with a grammatical representation [21]. GE has been used extensively for automatic design with promising results [22, 23, 24] motivating the exploration of its applicability for automatic CG [25, 26, 27] .

In our implementation of GE, the phenotype (a timeline) is a one-dimensional string

```
<timeline>::=<IEs><TLE_terminal>
<IEs>::=<IE><TLEs_more>
<IEs_more>::=<IE>|<IE><IEs_more>
<IEs>::= <IE_1>|<IE_2>|<IE_3>
<IE_1>::=gameplayEvent1(<Time1>)
<IE_2>::=gameplayEvent2(<Time2>)
<IE_3>::=gameplayEvent3(<Time3>)
<IE_terminal>::=endOfLevelEvent(0)
<Time1>::=[500,1000]
<Time2>::=[1000,2000]
<Time3>::=[2000,3000]
```

Figure 5.3 – *The design grammar employed to specify the levels' timeline.*

of interaction events (IEs). IEs are the possible events that can happen during a game session; this can be a jump in SMB, a rope cut or an interaction between the candy and a bump in CTR or swipe in Temple Run. Each event in the timeline is associated with a timestamp that specifies the exact time during the game in which this event is activated. Instead of using an absolute timestamp for each IE, we assign a number that indicates the timespan elapsed since the previous event was activated. The use of elapsed time permits the incorporation of context (game specific) information. For instance, the frequency of doing an action and/or the waiting time required after performing a specific action is game-dependent. In fast-paced games, such as Temple Run, the player has to rapidly perform swipes and tilts to avoid losing, while in other games, such as SMB or Candy Crush, the amount of time available for the player to perform an action is relatively long. The use of elapsed time also allows for action dependent time variant as some actions are followed by a longer waiting time than others. (Pressing an air-cushion in CTR is usually followed by a relatively long waiting time allowing the candy to reach a specific position, but shooting a gun in a FPS game is mostly followed by rapid reactions from opponents.) Finally, the use of the elapsed time allows for more efficient search since it reduces the size of our search space by eliminating a high number of invalid individuals with the exact or an overlapping timestamps.

Fig. 5.3 shows an abstract grammar employed by GE for timeline generation. The grammar is defined so that each level has at least two IEs and a terminal state (winning or losing the game). Each event can be one of the possible events in a game associated with an event-dependent timestamp.

The second part of the progressive approach is the use of a constructive method to evaluate the evolved timelines. Each timeline is passed on to a game simulator that progressively scans the timeline while building a compatible level design. A timeline is assigned a score by the simulator according to whether it could be matched with a playable design and to other heuristics that relate to aesthetics or design constrains.

5.2.2 Timeline Simulation

A timeline controls the order, the elapsed time and the type of the components that will be presented in a level. It does not however carry information about the components' specific properties such as their direction or their exact position in the game canvas. Moreover, the compatibility between the added components is not guaranteed (whether a specific action can actually be performed within a certain period of time from another). Thereafter, a generated timeline can not be guaranteed to be playable and therefore a timeline structure generated by the TLG needs to be evaluated. For this purpose, we use the game simulation layer. In this layer, each evolved timeline is mapped into its phenotype representation (a game level) while being simulated by an agent. Since we want our algorithm to be fast and reliable, and as the timelines contain missing information, our proposed approach to generate a matching design for a given timeline is to gradually construct the design as the timeline is being scanned, hence the word *progression*. More specifically, as the timeline is being simulated, components are added to the scene in a way that maximises the chance that the final design is playable according to the steps presented in Algorithm 2. Notice that we are aiming at generating playable designs but we consider playability as a minimal criterion and other factors might also be considered when assigning a score for a timeline.

Algorithm 2: The Progression Algorithm

```
   Data: E : list of events in the timeline with their associated timestamps;
   Set game timer T to 0;
   Start simulation;
1  while E contains more events, e do
2      if e is the End event then
3          return the level design;
4      if T = e.time then
5          create the associated component, c;
6          c.activate();
7      if T >e.time or lose then
8          return invalid;
```

The algorithm starts by converting the genotype (timeline) into a list of interaction events with their associated timestamps. The simulator then resets the game timer and starts the simulation. This is done by scanning the list of events and activating the top event in the stack when the game timer becomes equal to the event's timestamp (lines 4-6). Activating an event means placing the associated component in the game canvas (more details about how to intelligently do this later) and simulating the event's action (jumping over a gap or popping a bubble.) The simulation continues until the next event becomes activated (when the game timer reaches e.time) and this continues until the game termination event is reached (line 2) in which case the game is considered playable and the final design is returned.

Since the simulator can freely assign properties and positions to the components, it is likely that some of the configurations will fail to reach the termination event and consequently the timeline is considered invalid (line 7.) Therefore, the simulator is optimised so that it places the components *intelligently*; i.e. whenever a component is to be added, it is placed in a way that its position intersects with the trajectory of an agent playing the part of the game constructed so far. This way we can guarantee the existence of a path between the components. The termination event is also placed along this path, if possible, ensuring a playable design.

The informative placement of components solves only part of the problem, there remains the part where the characteristics of the components also play a role in whether a timeline is playable (this could be for example placing a gap that is too wide for the player to jump over). In this case, a valid timeline might be misclassified as invalid (a false positive case). This issue can be easily solved by repeating the simulation a number of times to allow the exploration of different configurations (note however that there is still a slight chance of misclassification).

Finally, the simulation might fail because the structure of the events in the timeline is indeed invalid (too short or too long time between the events or incompatible sequence of components (for example, a gap followed by stairs in SMB)) (line 7.)

5.2.3 Notes on the Progressive Approach

A core feature of our approach is that content is placed in a way that matches the timeline whilst guaranteeing playability (a timeline is always playable up to the point where the simulation fails). There is no need for any additional playability check afterwards. This is a main advantage over other CG methods proposed in the literature where two separated steps are usually employed to (1) generate complete designs and (2) performs a playability check [15, 9] which results in a significantly slower performance than the one obtained by the proposed approach.

Another vital contribution of the proposed framework is that it guarantees *usability*, i.e. all the components presented are necessary, and should be used, to complete the level, unlike previous attempts to generate content where this is not guaranteed [15, 9, 25]. This is an important issue in game design since the number of paths that could be followed to finish the game provides an estimate of difficulty. This issue, however, is more important in some games than others. The existence of extra components in SMB for example allows for level's segments with alternative paths, while in other games such as CTR that requires fine tuning of the components and their properties to preserve playability, the extra components are usually perceived as a design flaw.

Another important feature of the methodology followed is that the genotype-to-phenotype mapping is one-to-many, i.e. more than one playable design could be generated that match one evolved timeline (note that this could be the result of one or multiple runs of the algorithm). This is facilitated because of the imperfect information carried by the timeline which permits many successful interpretations. This is beneficial since it allows the designer to explore a number of alternatives or variants of the level that are structurally different but all share the exact in-game interactions.

Simulating a timeline indicates whether or not it could be mapped into a playable design. This forms the first step in evaluation. Other measures could also be considered in this step and depending on the designer preference, a fitness value will be returned indicating the "goodness" of the timeline.

In what follows, we demonstrate an application of our framework to a physics-based puzzle game to illustrate its applicability. We present the game and the modifications required to employ the method and we analyse the performance and the output obtained.

5.3 Customising the Framework for Cut the Rope

We use a clone of the game Cut the Rope as a testbed for our approach. The game has many characteristics and challenges that motivate choosing it as a testbed; the physics constraints applied and generated by the different components of the game necessitate considering factors when evaluating the content generated other than the ones usually considered for other game genres, testing for playability is another issue that differentiates this type of games since this needs to be done based on a physics simulator, finally, the game defines a new genre that has slightly been researched.

The gameplay consists of performing certain actions on specific components to redirect the candy towards Om Nom, a frog-like creature. Timing is an essential property of the game as the player has to perform specific actions at certain times to successfully finish the game.

There are many different level components in the original game and we use the basic ones in our clone: ropes, air-cushions, bubbles, rockets and bumpers. The possible actions that could be performed thereafter are: a rope cut, an air-cushion press, a bubble pop, a rocket press or a void action (not taking any action, i.e, waiting for the candy to reach a certain position).

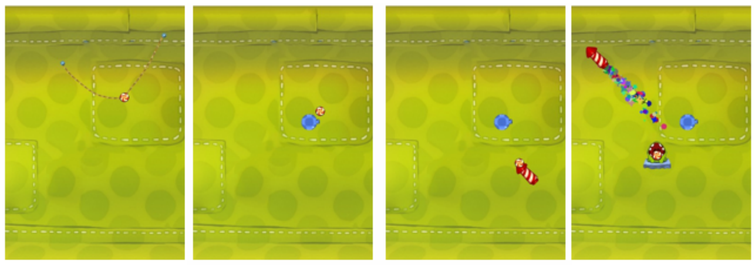
5.3.1 Timeline Generation

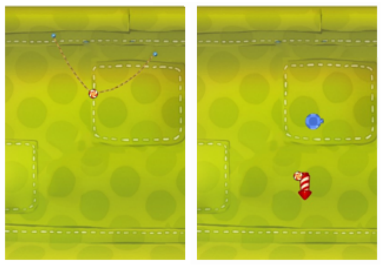
Customising this layer for CTR implies instantiating a new design grammar with the possible events in CTR and assigning the appropriate timestamps. The final grammar can be seen in Fig. 5.4 where the timespan values are assigned for each IE experimentally based on several evaluations to reflect the components' specific properties. An example timeline evolved using this grammar is: *rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(600) omNom_feed(0)*. This timeline consists of four IEs (two rope cuts followed by pressing an air cushion and finally pressing a rocket) and a game-end event.

```
<timeline>::=<IEs><IE_terminal>

<IEs>::=<IE><IEs_more>
<IEs_more>::=<IE>|<IE><IEs_more>
<IE>::=<rope_cut>|<aircush_press>|<bubble_pop>|<bumper_inter>|<rocket_press>
<rope_cut>::=rope_cut(<default_ET>)
<aircush_press>::=aircush_press(<default_ET>)
<bubble_pop>::=bubble_pop(<short_ET>)
<rocket_press>::=rocket_press(<short_ET>)
<bumper_inter>::=bumper_inter(<long_ET>)
<IE_terminal>::=OmNom_feed(0)
<short_ET>::=[600,1600]
<default_ET>::=[800,1800]
<long_ET>::=[1200,2200]
```

Figure 5.4 – *The design grammar employed to specify the levels' timeline in Cut the Rope.*


(a) *Progressively designing a successful mapping.*


(b) *Unsuccessful mapping.*

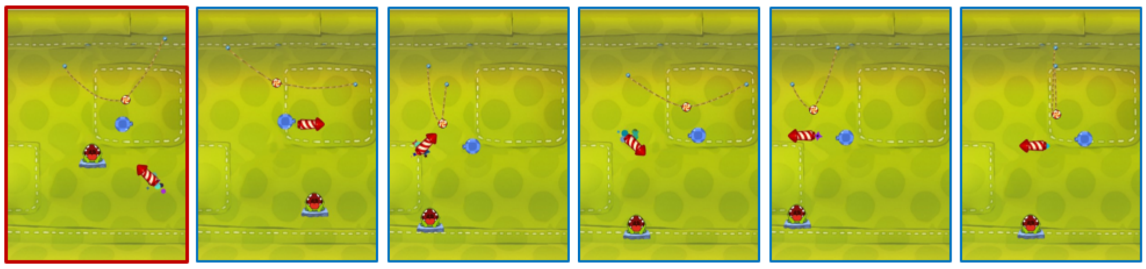

(c) *Examples of successful mappings for the same timeline to a number of dissimilar designs.*

Figure 5.5 – *A successful, unsuccessful and some example possible designs for the timeline: rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(700) omnom_feed(0).*

5.3.2 Timeline Simulation and Evaluation

Since CTR is a physics-based game, simulating the game requires a physics simulator that can handle the different physical properties presented in the game. The simulator, CRUST engine [19], thus is used to create designs that match given timelines following the algorithm presented in Algorithm 2. Fig. 5.5a presents the different steps followed by the simulator when mapping the timeline: *rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(600) omNom_feed(0)* into a game design. As can be seen, as the simulator starts scanning the timeline, two ropes attached to the candy are added with a very short timespan in-between. The simulator then activates the associated IEs by cutting the ropes. This initiates a candy free movement and when the time for an interaction with an air cushion is reached, the simulator adds an air cushion in a position that intersects with the candy's trajectory. The air cushion is directly activated and this leads to a slight change in the candy's path as the result of blowing air. The candy keeps moving downwards affected by its gravity and it hits the rocket that is created after 700 ms from activating the air cushion. Finally, the simulator adds Om Nom in the rocket's path and this termination event successfully ends the simulation resulting in a playable level.

As discussed earlier, the characteristics of the added components have a great impact on whether or not the final design is playable. In our previous example, adding a rocket directed downwards will result in losing the game and thereafter misclassifying the timeline as invalid (see Fig. 5.5b for illustration). Several runs of the simulation while differentiating the properties of the components will lead to many design variants. Some of which are indeed playable designs proving the validity of a timeline. Examples of possible playable levels for the timeline discussed previously can be seen in Fig. 5.5c.

The fitness function chosen to score the timelines is a weighted sum of several properties that capture different design, playability and aesthetic considerations. The list of features includes:

- The total duration of gameplay calculated as the sum of all elapsed times.
- Trajectory loops: the arrangement of the components on the canvas might lead to situations where the same position will be revisited by the candy. This is considered an inferior design as some of the components become obsolete.
- Overlapping components which occurs when the elapsed time assigned for two adjacent events is too short.
- Playability: in some cases, the simulator will repeatedly fail to map the timeline to a playable design.

Notice that the first condition defines a design constraint since levels that are too long or too short are likely to be uninteresting. The second condition depends on how the simulator chooses to place the components in the canvas and repeating the simulation might lead to different results. The final two conditions are actual faults in the timeline and thereafter they are given the highest weights. The exact values assigned for the weights given to each of the above conditions are experimentally chosen.

5.4 Implementation and Experimental Setup

The existing GEVA software [28] was used to implement the TLG tier. The experimental parameters are the following: 100 runs were initialised with the ramped half-and-half method, each run lasted for 30 generations with a population size of 10 individuals. The maximum derivation tree depth was set at 100, tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.9, and 3 maximum wraps were allowed. The termination criterion is creating the first valid design[1].

5.5 Results and Analysis

As our framework constitutes two modules, we will focus our analysis on the timeline generation and the design generation separately.

5.5.1 Timeline Analysis

In order to assess the quality of the generated timelines, their variation and the generation efficiency, we ran the TLG for 100 timelines and analysed the results of the valid timeline evolved. The results show that in most cases two generations are required to get a valid timeline which can be achieved relatively quickly (within 7.23 ± 9.19 sec) (note that this is mainly because of the efficiency of mapping the TL to a playable design). We also ran a simple analysis to investigate the difference in the length of the timeline evolved (the

[1]A video showing the implementation of the framework in CTR is available online: http://www.mohammadshaker.com/ropossum.html

number of IEs). The results show that the average length is 4.83 ± 0.82. The analysis also showed that more than 85% of the timelines are of length five or smaller. This tendency towards generating short timelines draws our attention to the design of our scoring scheme which does not account for the number of IEs. As a result, the system favours short timelines that could be easily mapped to valid designs.

To further investigate the generator's capabilities we calculated the number of occurrences of the different events in the timelines evolved. The results show that 48, 44, 47, 51 and 51 items are generated for *rope_cut, aircush_press, bubble_pop, bumper_inter and rocket_press*, respectively. The results approximate a uniform distribution demonstrating a good balancing capability.

5.5.2 Generator Expressivity Analysis

Visualising the distribution of the components helps us understand the simulator's capabilities and its expressive space. Colour maps is one of the methods used for this purpose [15]. This is done by generating a large amount of game content, converting each instance into a pixelated image and projecting all instances on a single image. To analyse our generator, colour maps are applied on individual components to ease the analysis and to give a better visualisation. Fig. 5.6 presents the colour maps obtained for Om Nom (assigned a green colour) in 100 levels generated from valid timelines. Note that Om Nom's placement corresponds to the position in the map where the end-game event takes place. The figure illustrates that a large portion of the canvas is explored.

The above visualisation helps use understand the diversity of levels generated from *different* timelines; given the two-step nature of the approach, it is also interesting to look at the diversity of levels generated from a *a single* timeline. For this purpose, the game simulator is set to run 100 times while trying to map the timeline *rope_cut(700) rope_cut(1000) bubble_press(1000) rocket_press(1500) OmNom_feed(0)* into playable designs. As a result, the simulator was able to successfully generate 63 playable designs while failing the rest of the attempts. Fig. 5.7 presents the distribution of rockets, ropes and Om Nom in the successful cases. The results illustrate a wide variety of structural differences for all items indicating that the same timeline can create numerous different levels, likely contributing to different types of player experience.

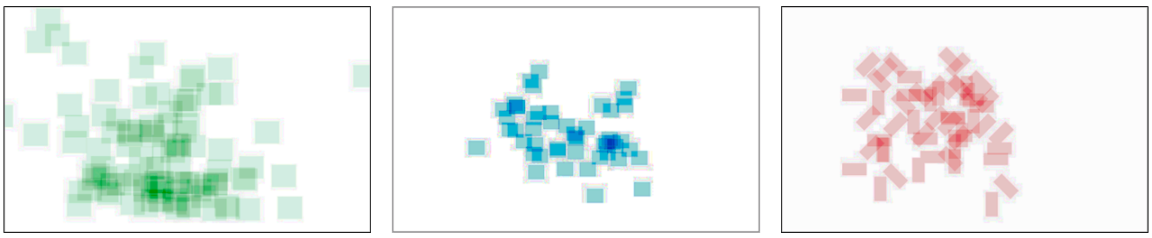
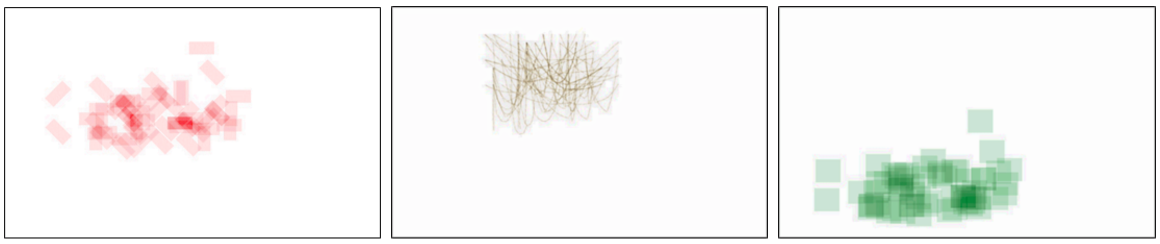


Figure 5.6 – *Om Nom's, blowers' and rockets' placement colour maps for 100 generated levels.*



(a) *Rockets* **(b)** *Ropes* **(c)** *Om Nom*

Figure 5.7 – *Colour maps for different components for 63 designs generated for the same timeline.*

5.6 Performance Comparison

In section 4.10 we discussed how the projection agent performed against the current state of the art. In this section, and in comparison to previous attempts to evolve playable content for the same testbed, we show how the new progressive approach shows superiority in terms of speed, variety, control and usability of generated content over other approaches (the projection agent and previous works in the same area [15, 9] don't have any control over the usability or the quality of the generated content for instance.) While it takes the simulation-based and the random agents 470 and 82 sec to evolve a complete playable level [15, 9], the proposed methodology is able to evolve timelines with more than one associated design in 9.79 sec while ensuring that all generated entities are used in all successful playthroughs. The time drops to 7.23 sec if we want to generate a level for only one associated design. Table 5.1 shows how our two new approaches perform against others in literature. Although the projection agent is the fastest for playability check of a given design, the progressive approach is the fastest for content creation. Also, the progressive agent is able to far excel in content quality (where all components are used and activated) with very feasible time for real-time application.

Table 5.1 – Comparison between the current approach and the previous ones. The table presents the time required to check if a design is playable (in seconds), the time required to evolve a playable design (in seconds) and the number of generations required for evolving a valid design.

Agent	*Playability*	$Evolution_{time}$	*Generations*
Pseudo-Random [15]	0.98 ± 0.64	21.27 ± 23.44	2.63 ± 1.82
Simulation-Based [9]	29.87 ± 58.28	470.1 ± 525.4	2.48 ± 1.58
Projection-Based	0.1 ± 0.02	13.1 ± 7.57	1 ± 0
Progressive	1.39 ± 0.87	7.23 ± 9.19	1 ± 0.9

5.7 Applicability of the Progressive Approach

Driven by the success of applying the progressive approach for the generation of content for CTR, we were very motivated to apply the progressive approach on different game genres. Therefore, we have applied it on our two newly designed games, namely, NEXT [29][2] and iNversion [30][3].

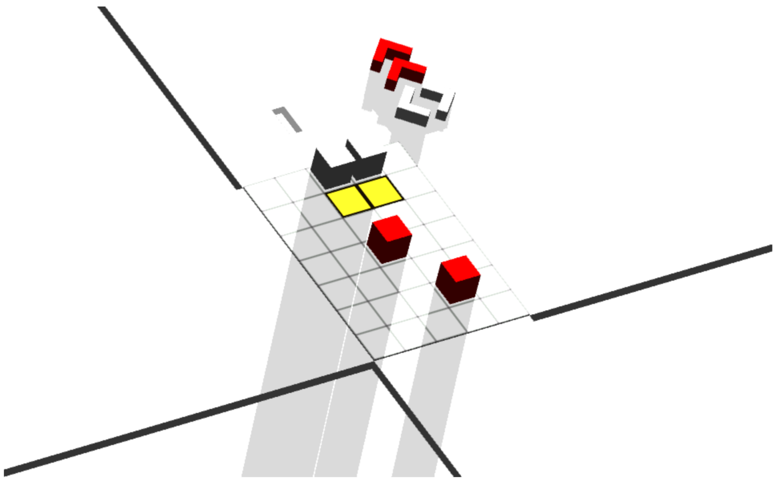


Figure 5.8 – *The progressive approach is employed for generating content for NEXT.*

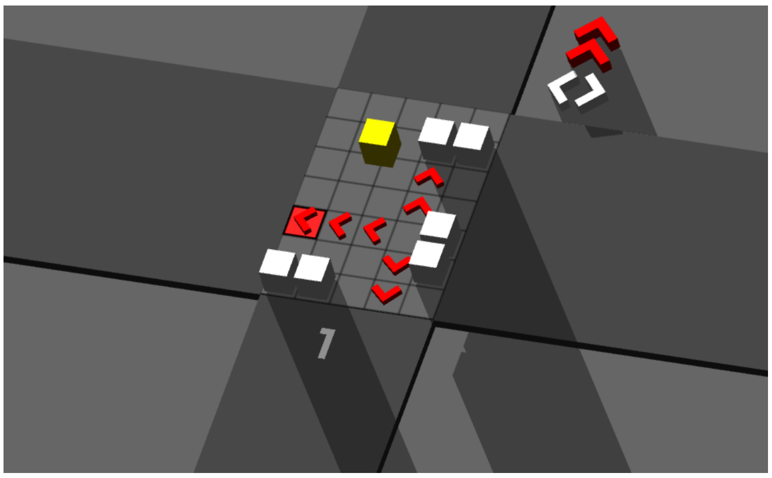


Figure 5.9 – *The progressive approach is employed for generating content for iNversion.*

NEXT and iNversion (see Fig. 5.8 and Fig .5.9) are 2D swipe games that have inverse game mechanics. Here, we will especially talk about NEXT since iNversion is similar. The main goal of NEXT is to lead the Red blocks to the Golden cells by swiping a gesture across an axis in a 7×5 canvas. Red blocks *glue* together when they are close and will move simultaneously together with the same swipe gesture. Only White blocks can stop the Red blocks from moving along an axis, serving as obstacles. As noted, this is a very different game to Cut the Rope. Yet, the progressive approach was easily customized to generate game content for NEXT.

In comparison to CTR, the grammar (in the TLG part of the framework) is very simple, specifying only the positions of the Golden cells and the number of moves required

[2]NEXT is available online: http:www.mohammadshaker.com/next.html

[3]iNversion is available online: http:www.mohammadshaker.com/inversion.html

by the Red cells to reach the Golden cells (fig. 5.10). The other part of the progression framework, the Timeline Simulation, is easily implemented as a reflection of the grammar itself, mapping the design grammar into an actual game content, simulating the moves of each Red cell specified in the grammars. A number of trials are given for the timeline simulation component in order to search for a valid design that fulfils the grammar. The process is the same as discussed in section 5.2.2.

```
<timeline>::=<golden_cells_origin><golden_cells>

<golden_cells_origin>::=origin(<x>, <y>)
<golden_cells>::=<goldin_cell>|<golden_cells>
<golden_cell>::=golden_cell(<moves>)

<x>::= [0..6]
<y>::= [0..5]
<moves>::= [1..10]
```

Figure 5.10 – *The design grammar employed to specify the levels' timeline in NEXT.*

5.7.1 Controlling Difficulty

In section 5.2.3, we have discussed that the difficulty of a level is somehow embedded in the grammar itself. Again, looking at the grammar in fig. 5.10, we can easily see that the simplest way to measure the difficulty is by controlling both; the number of Golden cells and the number of moves of each Red cell. This is done by controlling the fitness value assigned to the generated level design in the TLG part. The designer can define a preferred difficulty value and the TLG part will assign a fitness score for each generated level design according to the preferred difficulty value. In this way, we can see that the progressive approach framework can be easily expanded to handle the difficulty of the levels, generating levels of only preferred difficulty.

5.7.2 Final Remarks

This section emphasises our earlier claim that the progressive approach can be applied on different game genres. The progressive approach's main strength is that it's so intuitive that it can be applied across different games. Although, in this section, we have shown it's applicability on a different game genre (2D swipe) than the one studied in this thesis (physics-based puzzle), a more thorough analysis should be done to measure the approach's generated content quality, variety and performance across genres. This is a very interesting future direction that we will definitely take.

— 6 —
Conclusions and Future Directions

In this study we presented two methods for real-time assessment and content generation of games. First, we have proposed the Projection-based Approach. It is a method for visualising and checking playability for physics-based puzzle games. The method proposed considers the physical properties of different game components and accordingly defines their space of influence. Informative combination of the influence areas of all components in a given design defines the playable space in which interaction with the player can take place. This was effectively achieved through a careful consideration of the context information. The algorithm executes in a short time and provides informative visual feedback both in the offline mode when the agent is set to work on a complete design and in the online mode while the level is being designed. While the method differs considerably from all previous methods for game level analysis that we know of in that it is based on the novel concept of Projection Areas and search in projection space, it is still at its core an artificial intelligence method that performs search in a problem- and domain-appropriate space.

There are a number of physics-based puzzle games where we see the proposed approach easily applicable. We are thinking of games such as Amazing Alex, Sprinkle, iBlast, Moki, Enigmo 2, Touch Physics 2, Chalk Ball, and Angry Birds; these games have similar properties to Cut the Rope in terms of in-game entities and their movement and interaction. Generalising the method to work for any game of this genre, perhaps by encoding mechanics in a description language for physical puzzle games, would be exciting future work.

We then proposed a second, more generic method for content generation; the Progressive Approach. The framework is built on the idea that by combining the advantages of search-based and constructive approaches for content generation, a fast and expressive generator can be built. The generator is composed of two main components: an evolutionary based timeline generator and a constructive-based game simulator. Each timeline is sequence of in-game events that can be mapped to a game design by the simulator. Timelines are evolved using grammatical evolution and evaluated by the simulator which progressively scans a timeline and adds components when necessary while preserving playability. A timeline can be mapped to more than one design, facilitating the exploration of multiple levels with the same rhythm. The framework is tested in a physics-based game where timelines are evolved and scored according to playability and aesthetic constraints.

There are a number of interesting future directions: (1) The method shows promising results in our testbed and it would be interesting to validate its generality by applying it to games from other genres such as first-person shooters or endless runners. We have started working on this and the results are very encouraging. NEXT and iNversion are our two new game prototypes, available online, where the progressive approach framework is employed to generate levels for 2D swipe games, a completely different genre of the one discussed in this study (physics-based puzzle.) The progressive approach framework employed in NEXT and iNversion was able to generate different kind of levels of different difficulty. Studying the similarities between the generated content within the same game and between different games of different genres is an important future direction that can emphasise both the approach applicability and capability. (2) A measure of the game difficulty is somehow embedded in the definition of the timeline (the more the components and the shorter the time between the events the harder the level is). It would therefore be interesting to generate content of specified difficulty by modifying the calculation of the fitness. (3) One could also try to generate levels with multiple alternate timelines, that could be solved in different ways.

Acknowledgments

We thank ZeptoLab for giving us permission to use the original Cut The Rope graphical assets for research purposes.

Bibliography

[1] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation. *Applications of Evolutionary Computation*, pages 141–150, 2010.

[2] Havok, 2011. Havok, www.havok.com.

[3] Interactive Data, 2011. SpeedTree.

[4] G. Smith, J. Whitehead, and M. Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010.

[5] Adam M Smith. Open problem: Reusable gameplay trace samplers. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[6] Ruben Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 2. ACM, 2010.

[7] Antonios Liapis, Georgios Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of ACM Conference on Foundations of Digital Games*, 2013.

[8] Mohammad Shaker, Noor Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE). AAAI Press*, 2013.

[9] Mohammad Shaker, Noor Shaker, and Julian Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[10] Aaron Bauer, Seth Cooper, and Zoran Popovic. Automated redesign of local playspace properties, 2013.

[11] Ubisoft, Ubisoft Montpellier, and Digital Eclipse, 1995. Rayman , Ubisoft.

[12] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G.N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, pages 265–272. Citeseer, 2010.

[13] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[14] J. Ortega, N. Shaker, J. Togelius, and G.N. Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 2013.

[15] Mohammad Shaker, Mhd Hasan Sarhan, Ola Al Naameh, Noor Shaker, and Julian Togelius. Automatic generation and analysis of physics-based puzzle games. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[16] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. Constructive generation methods for dungeons and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[17] Mike Preuss, Antonios Liapis, and Julian Togelius. Searching for good and diverse game levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[18] Ian Millington. *Game physics engine development*. Morgan Kaufmann Pub, 2007.

[19] Shaker, Mohammad, 2013. CRUST, www.mohammadshaker.com/crust.html.

[20] A. Johnson. Clipper - an open source freeware library for clipping and offsetting lines and polygons, 2014. http://www.angusj.com/delphi/clipper.php.

[21] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[22] G.S. Hornby and J.B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 600–607. IEEE, 2001.

[23] J. Byrne, M. Fenton, E. Hemberg, J. McDermott, M. O'Neill, E. Shotton, and C. Nally. Combining structural analysis and multi-objective criteria for evolutionary architectural design. *Applications of Evolutionary Computation*, pages 204–213, 2011.

[24] M. O'Neill, J.M. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg. Shape grammars and grammatical evolution for evolutionary design. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1035–1042. ACM, 2009.

[25] N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, 2012.

[26] N. Shaker, G.N. Yannakakis, J. Togelius, M. Nicolau, and M. Oï¿$\frac{1}{2}$Neill. Evolving personalized content for super mario bros using grammatical evolution. 2012.

[27] J. Font, T. Mahlmann, D. Manrique, and J. Togelius. Towards the automatic generation of card games through grammar-guided genetic programming. *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 2013.

[28] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. Geva: grammatical evolution in java. *ACM SIGEVOlution*, 3(2):17–22, 2008.

[29] Shaker, Mohammad, 2015. NEXT, www.mohammadshaker.com/next.html.

[30] Shaker, Mohammad, 2015. iNversion, www.mohammadshaker.com/inversion.html.