Squeezer - A Mixed-Initiative Tool for Designing Juice Effects

Mads Johansen
madj@itu.dk
Center for Computer Games Research
IT University of Copenhagen
Denmark

Martin Pichlmair
mpic@itu.dk
Center for Computer Games Research
IT University of Copenhagen
Denmark

Sebastian Risi
sebr@itu.dk
Center for Computer Games Research
IT University of Copenhagen
Denmark

ABSTRACT

This paper presents a Mixed-Initiative version of *Squeezer*, a tool for designing juice effects in the Unity game engine. Drawing upon sound synthesizers and game description languages, Squeezer can synthesize common types of juice effects by combining simple building blocks into sequences. Additionally, Squeezer offers effect generation based on predefined recipes as well as an interface for interactively evolving effect sequences. We conducted a user study with five experts to verify the functionality and interest among game designers. By applying generative and evolutionary strategies to juice effect design, Squeezer allows game designers and researchers using games in their work to explore adding juice effects to their games and frameworks.

Squeezer is available at: https://github.com/pyjamads/Squeezer

CCS CONCEPTS

• **Applied computing → Computer games**; • **Human-centered computing → Graphical user interfaces**; Usability testing.

KEYWORDS

Game Development, Game Design, Juice Effects, Interaction Feedback, Toolkit, Prototyping, Generator, Interactive Evolution, Mixed-Initiative

1 INTRODUCTION

Game designers regularly create prototypes as part of their practice. Their purpose is to explore a design space, communicate game mechanics, and collect player feedback on game mechanics. While prototypes often require little to no juice – a game design term for visual and audible feedback and effects, see [24] – to verify if a rule or mechanic works, there are many cases where they depend on placeholder effects to provide enough feedback to make a game mechanic understandable by the player. In all cases, prototypes should be fast to make, which can be supported by tools.

Squeezer [18] assist game designers in prototyping by enhancing game feel [32] through simplifying the process of adding juice [19]. Squeezer can generate and link various juice effects to sequences. This way, designers can quickly and efficiently create rich feedback during the prototyping stage of game development.

In this paper, we expand upon the preliminary version of Squeezer by implementing category-based effect tree generation, mutation of effect trees, and adding an interface for the interactive evolution of effects. Interactive evolutionary computation [33] or interactive evolution builds on an AI-assisted creation loop. An interactive evolution system presents different artifacts to the user, who selects the artifacts they prefer among the selection. The system then generates a new set of artifacts through crossover and mutation. This next generation of artifacts is subsequently presented to the user, and so on. In practice, the user plays the part of the fitness function in the evolutionary process.

In our experiment, we explore if adding interactive evolution to Squeezer is beneficial for assisting designers in the exploration and design of effects for their game prototypes. Interactive evolution has been used to generate sonic and visual artifacts before, in tools such as Picbreeder [28], Endless Forms [7], MaestroGenesis [15], DrawCompileEvolve [36], ZzSsprite [9], or more recently in combination with deep generative methods [2]. It has not been applied to game feel design. In this paper, we present the results of testing two assumptions about the introduction of interactive evolution to Squeezer:

- Adding interactive evolution makes it easier for new users to get acquainted with Squeezer. It provides an alternative introduction to the possibilities of Squeezer[1] by letting users explore effect sequences interactively, based on examples, as opposed to manually testing individual effects one by one or familiarisation through documentation or code.
- Interactive evolution assists users in exploring the design space and in discovering new ideas for effects.

In summary, we show how generative and evolutionary strategies can assist designers and even surprise them by revealing unexplored areas of the design space when applied to juice effect design.

2 BACKGROUND

2.1 Game feel and Juice

Juice is the game design term for abundant feedback that amplifies interactions related to input and in-game events. The concept has gained a lot of traction over the last decade. Originally mostly discussed in the indie game community [19, 21], and in relation to prototyping [12], juice is nowadays being examined by a host of researchers [14, 20, 24]. Juice is superfluous from a strictly mechanical perspective but makes interacting with the system more pleasurable. Juice helps to sell the illusion that the game world has realistic properties, just as exaggeration in cartoons creates the illusion of life [34]. Hunicke states that "*Juiciness can be applied to abstract forms and elements and it is a way of embodying arbitrarily defined objects and giving them some aliveness, some qua, some thing, some tenderness.*" [16]

Juice is but one facet of 'Game Feel', a term that encompasses different kinds of polish of interaction design in games, a concept we reviewed thoroughly in [24]. The term itself was coined by Swink, who first wrote an article [31], and later a book [32] about this

[1]Further supported by a small suite of demo scenes, with effects already implemented, that new users can poke and prod at.

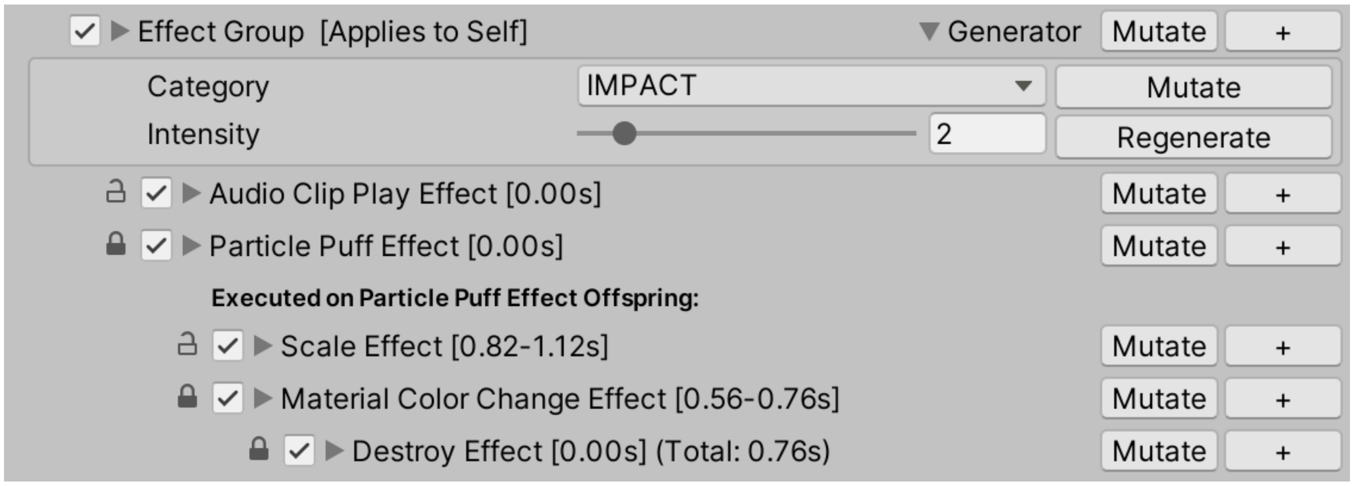

Effect Group [Applies to Self]
Generator
Mutate
+
Category
IMPACT
Mutate
Intensity
2
Regenerate
Audio Clip Play Effect [0.00s]
Particle Puff Effect [0.00s]
Executed on Particle Puff Effect Offspring:
Scale Effect [0.82-1.12s]
Material Color Change Effect [0.56-0.76s]
Destroy Effect [0.00s] (Total: 0.76s)

Figure 1: The effect tree generator interface featuring a drop-down menu for selecting a category and a slider for determining the intensity. The mutate and regenerate buttons respect locked parts of the effect tree.

topic. Fullerton describes polishing as "*the impression of physicality created by layering of reactive motion, proactive motion, sounds, and effects, and the synergy between those layers*" [11], an observation remarkably in line with Hunicke's description of 'juiciness'.

Both game feel, and juiciness have been widely discussed in the game development community [24, 30]. Developers like Jonasson & Purho [19] as well as Nijman [21] examine juice and game feel from their respective points of view, offering insights into how crucial this aspect of game design is to practitioners. Swink's book, and the talks by Jonasson & Purho and Nijman, are the primary sources for introducing the concepts of game feel and juiciness. Although all three had excellent demos available when they were released, these resources have unfortunately since deteriorated or disappeared.

New projects, such as MMFeedbacks[2] [10] and Game Maker's Toolkit [3, 4], continue to offer practical and analytical tools for designing game feel. MMFeedbacks is an expert tool for adding juiciness to Unity games. MMFeedbacks handles triggering, delaying, and sequencing effects yet subsidizes designing some of them to various existing subsystems in Unity. For instance, the user still needs to understand the Unity particle system and create the desired effect with that system before integrating it into MMFeedbacks. This design choice makes sense for MMFeedbacks as an expert tool because the Unity subsystems (e.g., Particle system, Cinemachine, Timeline, and Animator) are well suited for their specific purposes. However, an inexperienced user or a designer sketching out part of a game might not have the skills or the time to use MMFeedbacks meaningfully.

2.2 Procedural Content Generation

Procedural Content Generation (PCG) is a collection of methods to automate content generation for games [29]. Instead of manually designing game elements, PCG systems implement design rules for their automatic creation. PCG can be found in a wide variety of areas of asset production. Hendrikx et al. [13] map the field in their survey paper and provide an overview of common applications. Modern game engines like Unity support some procedural content generation systems out of the box (for example, terrain generation), but most PCG systems have to be either acquired or custom-built for each game. It is important to note that there are run-time and design time (offline) applications of procedural content generation. Often resource-intensive algorithms are run at design time. If curation is a part of the design process, the system also needs to be run at design time. An example of a tool working like this is SFXR, a sound effect tool [23].

SFXR has been used for a long time to lift the appeal of prototypes and game jam games [12] via "good enough" placeholder sound effects. Pettersson originally developed SFXR in 2007 for game jam participants to "*provide a simple means of getting basic sound effects into a game for those people who were working hard to get their entries done within the 48 hours.*" [23]

SFXR is a procedural content generator also known as a synthesizer. It is operated simply by clicking a button with a category name and pressing it repeatedly until the user hears a desirable sound effect. Apart from the sound generation, the user can also manually tune each of the more than twenty different parameters or mutate all parameters. The categories read as follows: [Coin/Pickup, Laser/Shoot, Explosion, Power-up, Hit/Hurt, Jump, Blip/Select, Random]. The categories act as presets, explicitly setting some parameters, limiting other parameters to specified ranges, and randomizing the rest. The sounds generated by SFXR can then be inserted as placeholders in a game or prototype until it is mature enough to get a sound designer involved or a sound pack integrated. For a game designer, applying placeholders can often reveal which mechanics they can build on and which they should remove. Designers often have to generate and try out 10-20 different versions to arrive at a sound effect that works in a particular situation. An interface that supports rapid iteration can thus speed up design significantly.

Squeezer supports rapid iteration by combining generative properties like those exhibited by SFXR, the simplicity of effect design

[2]https://feedbacks.moremountains.com/

such as in Particle FX Designer[3], and the universality of a feedback system like MMFeedbacks. Additionally, mutation and interactive evolution allow Squeezer to function as an effect sequence exploration tool.

3 IMPLEMENTATION

Prototyping is the phase of game development where iteration happens the fastest. The game modeled in the prototype is usually small – a single game mechanic, rule, or interaction. Prototyping is part of the exploration phase of design and is used to gain insights into the design space. Squeezer supports this exploration with the assistance of interactive evolution. This mechanism can be used to narrow down the possible design solutions or to arrive at new ideas. Squeezer has the power to surprise the user and lead them to design inspiration.

To encourage practical application and get access to high-quality test cases, Squeezer works with the widely used Unity[4] game engine. Unity lets users quickly develop and integrate custom tools to extend their editor. These tools are also sold commercially via the Unity Asset Store[5]. Users are accustomed to adding extra libraries to their projects to extend the functionality of the game engine and its editor. By developing Squeezer for Unity, we increase the probability of real-world applications and the number of potential users. However, the core functionality of Squeezer does not require Unity, and the underlying open source project can be adapted to other game engines.

3.1 Core Functionality

The three core elements of Squeezer are (1) triggering effects, (2) effect sequencing, and (3) effect execution. Effects are triggered by a simple event system that ties into the prototype's code or game events. The section about integration describes how to achieve that.

Effects are sequenced by structuring them into a tree and by using relative time offsets (delays) as seen in Fig. 3. Each node in the tree will execute all child nodes upon completion. The tree is displayed in its collapsed form, similar to the description language VGDL [27], to increase readability and simplicity. Effect nodes can be expanded to view and adjust their properties. They can also be mutated randomly to explore alternate variations of an effect quickly.

The executor manages the execution of effects and is in charge of scheduling and continuously updating active effects after they are triggered. Effects track their progress internally, letting the executor know when they finish and can be removed from active effects. When an effect ends, it will clean up after itself and queue all subsequent effects with the executor.

The executor is implemented like a simple tweening [5, 25] system, designed to handle one-shot as well as durational effects. Tweening systems are commonly used to drive simple animations and effects within game projects. Most tweening system implementations leave the interpolation to the individual tween classes, like Squeezers executor, which calls the active tweens on every update.

[3]https://codemanu.itch.io/particle-fx-designer
[4]https://unity.com/
[5]https://assetstore.unity.com/

Yet Squeezer is more than a tweening system: the effect sequence descriptions allow run-time manipulation and fast iteration on ideas. The interactive evolution interface allows exploration of the range of possible effects and combinations with very little knowledge of how the system works. The more advanced Inspector UI (see Fig. 3 and 1) can be used manually or in conjunction with interactive evolution, providing additional control and manual manipulation of values. Additionally, with the export and import of full or partial descriptions, users can easily create a library of effect sequences and apply them widely in a project or between projects. Squeezer also includes complex effects such as the SFXR[6] [23] audio synth.

3.2 Integration

In addition to the core elements, Squeezer provides several layers of user interfaces and code interfaces (Application Programming Interfaces or APIs) that allow designers and game programmers to integrate the tool exactly where and how they want. A setup window to help setting up the framework (step one in Fig. 2), where the user can select game objects or tags and the event triggers they would like effects to respond to. The inspector interface, with its simplified tree view providing a simple visual interface to generate, mutate, add and remove elements from effect sequences (step two in Fig. 2). As well as the expanded effect editor (step three in Fig. 2) that can be used to modify effect parameters. For those who prefer an API, Squeezer allows triggering and saving effect sequences generated by the sequence generator with the shorthand *Squeezer.Trigger(...)*. This shorthand generates an effect sequence based on a selected category and triggers it with the provided target, position, optionally a direction, and handing back a reference to the sequence for reuse later on. Programmers can also build and trigger effects or sequences manually; examples of this can be found in the sequence generator and several of the spawner effects (see Section 3.5).

3.3 Workflow

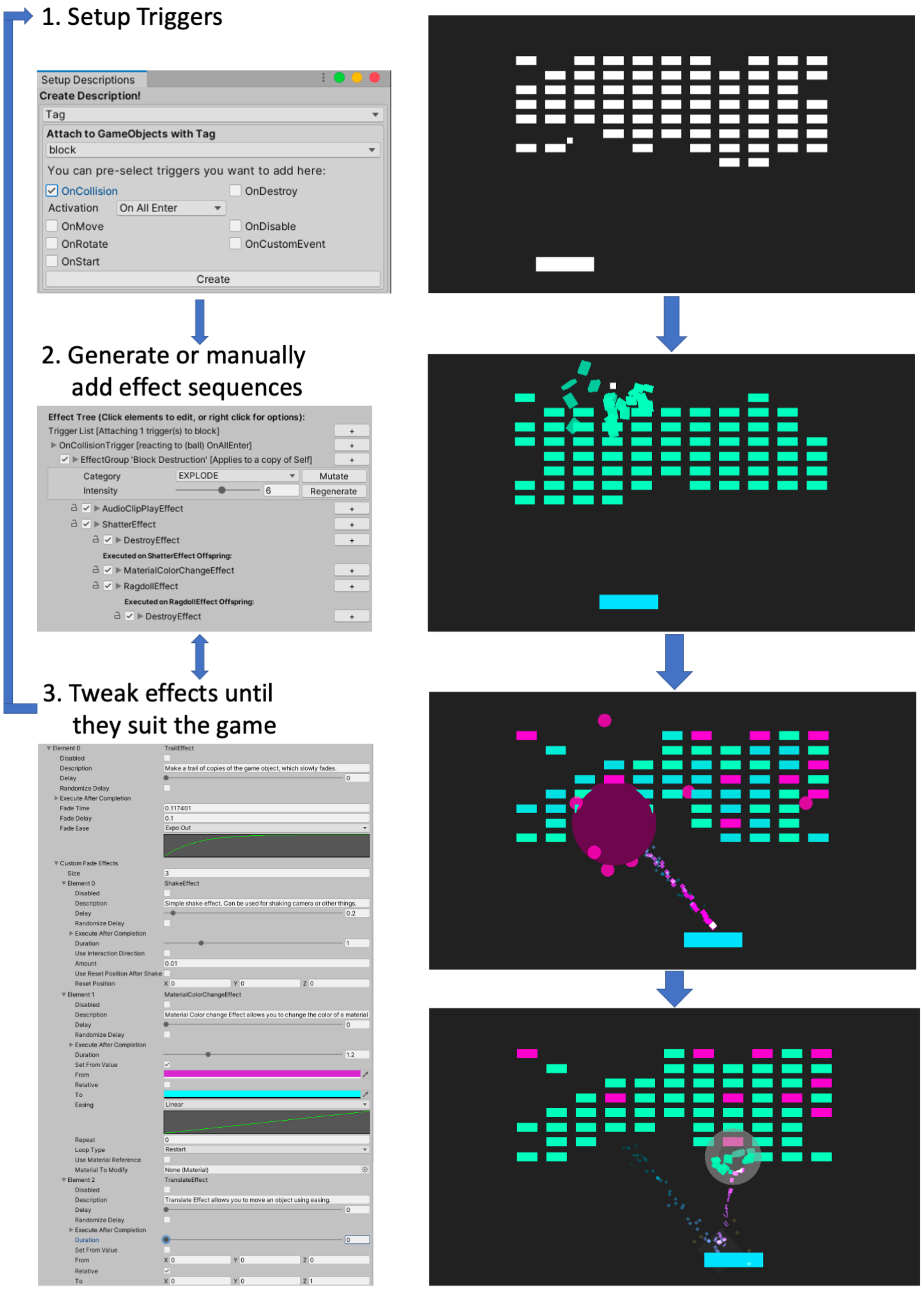
Fig. 2 shows a potential workflow for Squeezer, along with juice iterations on a breakout clone. Step one shows the setup window and basic game without effects. After setting up the triggers, step two shows the game with a bit of color and an initial generated effect sequence for block destruction. Step three shows the foldout menu for editing parameters of the trail effect. On the right side of step three, you can see a few iterations of the effects. Designers work iteratively, going back and forth between steps two and three, in addition to repeating steps one to three for every class of objects and event triggers they need.

3.4 Demo Scenes

We created four demo scenes in 2D and 3D. The demo scenes trigger effects on simple game objects when specific events occur. They are named Timer, Move, Jump and Shoot according to the event they demonstrate. All scenes either feature 2D objects and an orthographic camera or 3D objects and a rotating perspective camera. In Fig. 5) you can see one of the 3D demo scenes in the middle of executing an explosion effect.

[6]The Unity port of SFXR called usfxr: https://github.com/zeh/usfxr

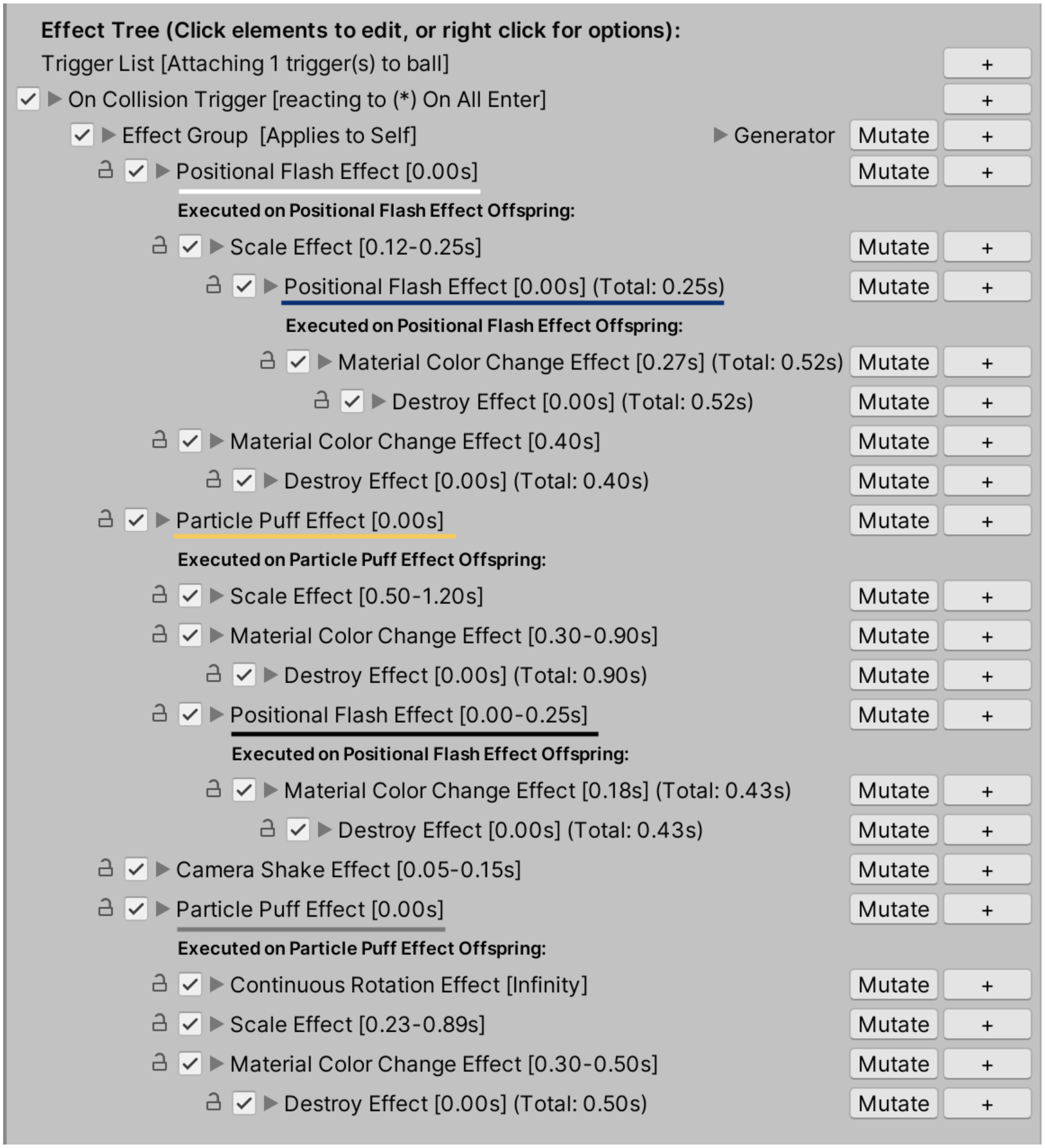
-70pt-


1. Setup Triggers
2. Generate or manually add effect sequences
3. Tweak effects until they suit the game

```
Setup Descriptions
Create Description!
Tag
Attach to GameObjects with Tag
block
You can pre-select triggers you want to add here:
[✓] OnCollision                 [ ] OnDestroy
Activation    On All Enter
[ ] OnMove                      [ ] OnDisable
[ ] OnRotate                    [ ] OnCustomEvent
[ ] OnStart
Create
```

```
Effect Tree (Click elements to edit, or right click for options):
Trigger List [Attaching 1 trigger(s) to block]
▶ OnCollisionTrigger [reacting to (ball) OnAllEnter]
  ✓ ▶ EffectGroup 'Block Destruction' [Applies to a copy of Self]
      Category    EXPLODE          Mutate
      Intensity   6                Regenerate
      ✓ ▶ AudioClipPlayEffect
      ✓ ▶ ShatterEffect
          ✓ ▶ DestroyEffect
          Executed on ShatterEffect Offspring:
          ✓ ▶ MaterialColorChangeEffect
          ✓ ▶ RagdollEffect
              Executed on RagdollEffect Offspring:
              ✓ ▶ DestroyEffect
```

```
▼ Element 0                        TrailEffect
    Disabled                       [ ]
    Description                    Make a trail of copies of the game object, which slowly fades.
    Delay                          0
    Randomize Delay                [ ]
  ▶ Execute After Completion
    Fade Time                      0.117401
    Fade Delay                     0.1
    Fade Ease                      Expo Out
  ▼ Custom Fade Effects
      Size                         3
    ▼ Element 0                    ShakeEffect
        Disabled                   [ ]
        Description                Simple shake effect. Can be used for shaking camera or other things.
        Delay                      0.2
        Randomize Delay            [ ]
      ▶ Execute After Completion
        Duration                   1
        Use Interaction Direction  [ ]
        Amount                     0.01
        Use Reset Position After Shake [ ]
        Reset Position             X 0   Y 0   Z 0
    ▼ Element 1                    MaterialColorChangeEffect
        Disabled                   [ ]
        Description                Material Color change Effect allows you to change the color of a material
        Delay                      0
        Randomize Delay            [ ]
      ▶ Execute After Completion
        Duration                   1.2
        Set From Value             [✓]
        From
        Relative                   [ ]
        To
        Easing                     Linear
        Repeat                     0
        Loop Type                  Restart
        Use Material Reference     [ ]
        Material To Modify         None (Material)
    ▼ Element 2                    TranslateEffect
        Disabled                   [ ]
        Description                Translate Effect allows you to move an object using easing.
        Delay                      0
        Randomize Delay            [ ]
      ▶ Execute After Completion
        Duration                   0
        Set From Value             [ ]
        From                       X 0   Y 0   Z 0
        Relative                   [✓]
        To                         X 0   Y 0   Z 1
```



Effect Tree (Click elements to edit, or right click for options):

```
Trigger List [Attaching 1 trigger(s) to ball]
✓ ▶ On Collision Trigger [reacting to (*) On All Enter]
  ✓ ▶ Effect Group  [Applies to Self]
    ✓ ▶ Positional Flash Effect [0.00s]
      Executed on Positional Flash Effect Offspring:
      ✓ ▶ Scale Effect [0.12-0.25s]
        ✓ ▶ Positional Flash Effect [0.00s] (Total: 0.25s)
          Executed on Positional Flash Effect Offspring:
          ✓ ▶ Material Color Change Effect [0.27s] (Total: 0.52s)
            ✓ ▶ Destroy Effect [0.00s] (Total: 0.52s)
      ✓ ▶ Material Color Change Effect [0.40s]
        ✓ ▶ Destroy Effect [0.00s] (Total: 0.40s)
    ✓ ▶ Particle Puff Effect [0.00s]
      Executed on Particle Puff Effect Offspring:
      ✓ ▶ Scale Effect [0.50-1.20s]
      ✓ ▶ Material Color Change Effect [0.30-0.90s]
        ✓ ▶ Destroy Effect [0.00s] (Total: 0.90s)
      ✓ ▶ Positional Flash Effect [0.00-0.25s]
        Executed on Positional Flash Effect Offspring:
        ✓ ▶ Material Color Change Effect [0.18s] (Total: 0.43s)
          ✓ ▶ Destroy Effect [0.00s] (Total: 0.43s)
    ✓ ▶ Camera Shake Effect [0.05-0.15s]
    ✓ ▶ Particle Puff Effect [0.00s]
      Executed on Particle Puff Effect Offspring:
      ✓ ▶ Continuous Rotation Effect [Infinity]
      ✓ ▶ Scale Effect [0.23-0.89s]
      ✓ ▶ Material Color Change Effect [0.30-0.50s]
        ✓ ▶ Destroy Effect [0.00s] (Total: 0.50s)
```

Figure 3: The description for an Explosion effect sequence. Underlines added externally to associate descriptions and visuals with the animation seen in Fig. 4.

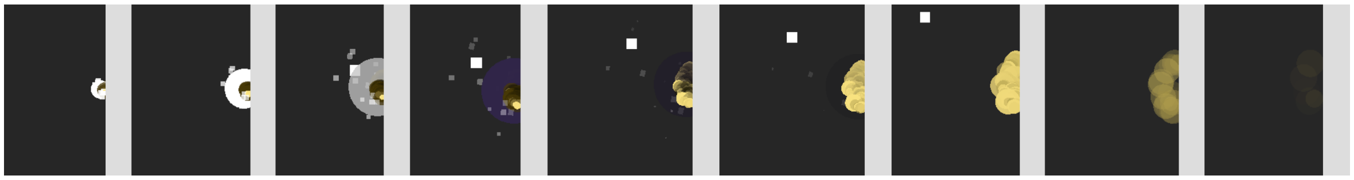


Figure 4: The "synthesized" effect shown at 5fps by executing the description in Fig. 3. The big white square is the breakout ball (i.e. not part of the effect). There are three visible parts of the effect sequence in the description (the “Camera Shake” is not visible here). The first is the “Positional Flash” (white underline in Fig. 3) which shows up as a white circle in the first three frames, scaling up and fading out. After the scale up finishes a blue flash (blue underline in Fig. 3) appears and fades out. The second part of the sequence is the “Particle Puff” (yellow underline in Fig. 3) controlling the slowly expanding yellow circles, each will spawn a quick black flash (black underline in Fig. 3), giving the slight illusion of smoky dust as they expand and fade away. The third part of the sequence is the “Particle Puff” (gray underline in Fig. 3) simulating debris with the small fast moving gray boxes.

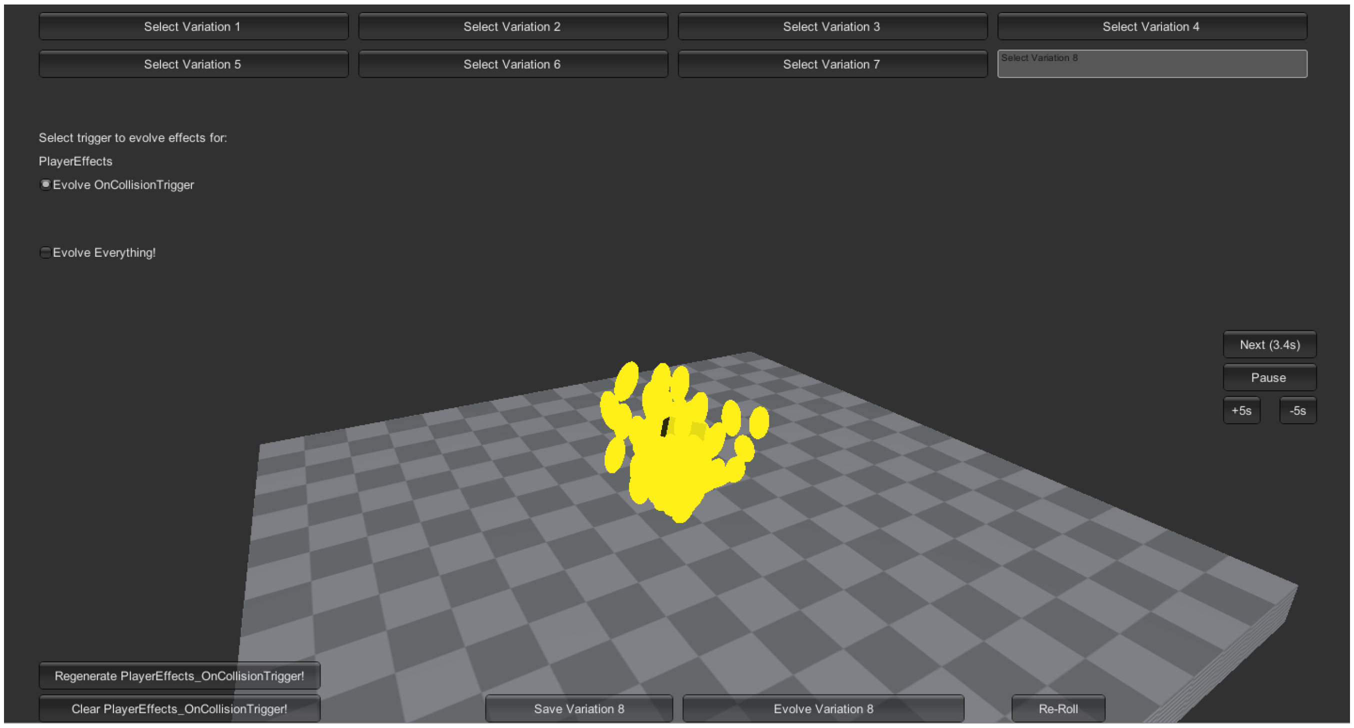


Figure 5: The run-time interactive evolution interface, seen here showcasing an explosion effect in a 3D scene.

Our original demo scene [18], a breakout clone, mainly consisted of events triggered by collisions. These new scenes revealed issues with trigger data as the initial version of Squeezer only passed a single 3D vector. The particle effect called “ParticlePuffEffect” provided the perfect test case for the amount of control and data needed when triggering events in the new demo scenes. The particle puff effect requires two parameters, a position, and a direction. Without changing the data provided by the triggers, the effect would only get a direction and would be forced to use the target object’s position. However, both in the Jump and Shoot demo scenes, triggering particles at the center of a game object would be incorrect. For events such as collisions, the center of either colliding object would also not be an accurate position to spawn an effect. While technically feasible for a designer to add an invisible game object as the trigger target[7], allowing the system to pass a relative position offset removes the need to do that. Triggers can now provide effects with “TriggerData” specifying both direction and position, allowing the particle puff effect to accurately execute at the location of the impact between two objects. The particle puff effect itself provides a simple way to create a small cloud of particles, expanding in a selected pattern.

The demo scenes provide an easy way to design and test effects in isolation. However, a more realistic testing scenario presented itself as the user study was about to begin. The developers of DISC

[7]This is indeed a well known “hack” designers exploit to implement position offsets visually in Unity

ROOM [6] launched a game jam, where participants could create their different versions of the game. As part of this promotional event, a tutorial video for a simple disc room prototype with accompanying source files, sprites, and sound pack, was released[8]. Squeezer includes this simple DISC ROOM prototype, re-created in Unity with the sprite and sound assets.

3.5 Architecture

The initial problem of representing and triggering effects based on events or interactions has been solved many times. One solution is a hierarchical description approach, including an ontology. This approach is found in PyVGDL, JavaVGDL, and UnityVGDL [17, 22, 26]. VGDL frameworks execute entire games based on this structure and effectively hide complexity in the descriptions with the provided ontology. Both, Squeezer's data structure and its textual representation of effect sequences use this hierarchical description approach. Juice effect sequences require scheduling, both sequentially and simultaneously, which complicates the nesting logic.

The data structure in Squeezer is hierarchical and simplifies the complexity of triggers and effects by offering classes for specific purposes. Broken down from root to leaf, the data structure is as follows (the hierarchy can be seen in Fig. 3, in a partial view of the Description inspector):

- Description – in charge of attaching triggers to actual game objects in the game, contains a list of Triggers (The outer level containing the "Effect Tree" in Fig. 3).
- Triggers – define which events cause effects to occur. Selected from seven predefined trigger types. Contains a list of effect groups (In Fig. 3, only one trigger is attached to the description, namely "On Collision Trigger").
- Effect Groups – determine which game objects the effects will be executed on and contain lists of effects (In Fig. 3 only a default "Applies to Self" effect group has been added to the collision trigger).
- Effects – include functionality to execute the effects themselves and a list of effects to apply on completion. Effects that spawn objects, such as particle puff, additionally contain a list of effects to run on any generated objects. (In Fig. 3 we see four effects on the root level, two different particle puffs, a positional flash, and a camera shake. Each of those, in turn, contain effects to be executed on either their "spawned" offspring or on their target as they complete).

Effect trees represent sequences. Effects on the same level – with the same parent – will be queued simultaneously. Nested effects execute after the parent has completed. In addition to this, a delay can be added to effects, offsetting them from other effects on the same level. Delays are handled internally by the effects themselves and begin counting down as soon as the effects are triggered.

In [18] we selected effect triggering events by analysing industry talks [19, 21]. This allowed us to identify an initial set of event triggers:

- OnStart, triggered when creating an object/when the game starts.
- OnCollision, triggered when a collision occurs.
- OnMove, triggered while moving or changing movement state.
- OnRotate, triggered when the object rotates in some way.
- OnDestroy, triggered when destroying an object.
- OnDisable, triggered when an object becomes disabled (a common way of "destroying" objects, without invoking garbage collection in Unity)
- OnCustomEvent, triggered when the system receives a custom event (e.g., named custom events such as Shoot, Jump or FadeInComplete)

Since a lot of games rely on Finite State Machines (FSMs) to control game logic, we added a new trigger recently that is called **OnStateChanged**. It is capable of tracking and activating based on changes to specific script parameters (like FSM states), as well as of checking simple conditions. The trigger detects when the value of observed parameters changes (e.g., when the state of a character changes from OnGround to Jumping or when the player's speed exceeds a specified value). Reacting directly to changes in the game state non-invasively can be an excellent alternative to adding code for triggering custom events in multiple places.

Although triggers are part of the hierarchical data structure, each trigger has an accompanying detection script attached to game objects in the game. These detection scripts react to game events and activate their associated effect groups. The effect groups determine the targets of the effect sequences defined within them. Usually, the effect groups only target the triggering object. Yet, sometimes they can help make other game objects seem to respond to specific events[9].

The main groups of effects are; sound effects, color effects, particle and trail effects, transform effects (translate, rotate, scale), time dilation effects, flashing effects (full-screen or localized), shake (quick random translations), and wiggle (rotational shake) effects. However, common for all effects is that they can be delayed, applied to various targets, sequenced (scheduled in relation to other effects), and executed dependent or independent of in-game timescale. Apart from those common properties, feedback effects tend to be durational. The most common are tween effects. The word tween comes from "inbetweening" [25], which originated in cartoon animation, where a senior would draw keyframes of animation sequences, and juniors would then fill in the timelines between those keyframes. A tween interpolates between a beginning and end value over a duration. The interpolation can be linear or eased in and out using easing curves[10]. The easing functions simulate acceleration and deceleration when starting and stopping, based on various functions like a sine wave or an exponential function. Another class of effects is those that spawn additional objects in the game instead of manipulating objects that already exist. Trails and other particle effects that generate objects contain a separate list of effects executed on their offspring. The four types of effects are *One shot, Durational, Tween, and Spawner effects.*

3.5.1 Extending Functionality. Squeezer is open source and can easily be extended with custom effects. Those can be based on one

[8]DISC ROOM JAM Tutorial, available at https://youtu.be/Dtt5X7twhxA

[9]This can be seen in the Squeezer showcase [available at: https://github.com/pyjamads/Squeezer/tree/master/Showcase] when all blocks in breakout react to the destruction of a single block

[10]see examples on http://easings.net/

of four effect types – one shot, spawning, durational and tween – depending on what is appropriate. Adding new effects to the system is as easy as copying or inheriting other effects, changing the name and performed logic, and saving the file. Squeezer uses reflection[11] to show available effects. The inheritance structure and serialization allow the system to display and execute the effects correctly.

Similarly, new triggers can be added. That process is slightly more complicated, as the detection scripts need to be created and the attaching logic has to be implemented as well.

It should be noted that spawning effects – Trail, Shatter, Copy and Particle Puff – all instantiate regular Unity game objects, which makes them very versatile. The objects in question can be custom-made for this purpose by the designer. Creating a large amount of game objects in Unity is resource-intensive, so game performance has to be monitored when effects spawn objects.

3.6 Content Categories

In [18] we identified and proposed the idea of generating effect sequences based on different categories. We started out with SFXR's eight categories of effects (see above) but removed Blip/Select, Coin/Pickup and Power-up. We limited the Squeezer categories to a set of simple arcade game mechanics, that suited our test cases. We added two new options: Player Move and Projectile Move. Meant for continuous triggers and have no counterpart in SFXR. Finally we renamed some of them to make their use clearer. The set of categories is a starting point for effect sequence generation, and we expect this list to be expanded in the future. In the end we ended up with the following content categories (SFXR name on the left, Squeezer name on the right):

- Random $\longrightarrow$ Random
- Explosion $\longrightarrow$ Destroy/Explode
- Jump $\longrightarrow$ Jump
- Laser/Shoot $\longrightarrow$ Shoot
- Hit/Hurt $\longrightarrow$ Impact
- N/A $\longrightarrow$ Projectile Move
- N/A $\longrightarrow$ Player Move

Fig. 1 shows the generator interface, where the category and intensity can be selected. The intensity value controls size, severity and sound volume, on a scale between one and ten (where one is the least intense). The intensity scale is split internally into four levels, [1-3] is low intensity variations, [4-6] is medium intensity, [7-9] is high intensity, and [10] is extreme intensity. The different intensity levels, have slightly altered effect sequences for some categories, but otherwise scale parameters linearly.

3.7 Generation, Mutation and Evolution

We handcrafted base sequences for each category. These sequences are the foundation of the generator. In the generator effects are initialized with parameters randomized within predefined ranges. This allows Squeezer to generate distinct effects with reasonable initial values.

Additionally, the sequences themselves are mutated at initialisation by randomly adding and removing effects in the sequence. The idea is that even significant mutations maintain some traits from the base sequence of the specified category. The logic for mutating effect trees combines genetic programming's [1] tree mutations with effect parameter mutations. This means that when mutating the effect sequence, a control parameter is used as a probability measure for adding or removing effects to the tree. While mutating each individual effect, the control parameter controls the probability that a value changes and how much it should be adjusted.

However, Squeezer is a design tool, and that means having a designer in the loop to determine fitness and guide the mutations along. To assist in this process, the user interface offers a locking mechanism[12]. Locked effects will not be mutated or removed while the rest of the tree gets mutated. This allows designers to 'freeze' parts of the effect sequence they like while evolving the remaining tree (one-armed bandit style). Designers can use this functionality and repeatedly mutate selected parts of the effect tree, gradually homing in on a final result.

3.8 User Interface

Squeezer offers interactive evolution through a run-time interface as well as through an inspector interface that is integrated into the engine's editor. In both instances, the genotype is a tree structure of effect class instances and the phenotype is the resulting sequence of "synthesized" effects that appear in the game.

The run-time interface consists of four control areas, presented as an overlay on top of the game view, as seen in Fig. 5. The top area lists each individual in the population (default is eight), and which one is currently selected. The left area lists the various triggers available for evolution. Each trigger is evolved separately, in order to avoid information overload. However, the last option on the list does allow bold designers to evolve everything at once. The bottom area contains buttons for starting over, saving the current individual, evolving the current individual or re-rolling the evolution from the previous stage. Both, saving and evolving, cause the current selected individual to be bookmarked, so that a designer can return to any step in the evolution they selected or saved at a later time[13]. The flow controls are found on the right side. These handle the timer that automatically switches between different individuals in the population. The interface shows only one individual at a time, and the game is reloaded when switching between them. Reloading the game scene, functions as a "palate cleanser", clearing out any lingering settings, colors or game objects the previous individual may have left behind. An individual can be skipped quickly, the display time can be changed and the timer can be paused entirely, allowing a designer to study the effects at the pace they see fit.

The inspector interface, that we integrated into the Unity editor, enables mutation of each effect group or individual effects. It can be used on its own or in combination with the run-time interface. In the latter case, locked and disabled parts of the effect tree will get copied to the next generation of individuals, guiding the evolution. The inspector interface allows the mutation of individual effects in the tree, and of entire effect groups. Effects can be reordered, deleted and copied between different parts of the effect tree. Effects

[11] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

[12] The padlock icons in Fig. 1 indicate which elements are locked

[13] These bookmarked descriptions have to be loaded back individually and manually through the inspector interface on the specific description game object. A workflow that should be improved in the future.

in an effect group can be mutated and regenerated from a selected category and with a specified intensity.

4 USER TEST

We invited five experienced game designers, two identifying as female and two identifying as male, to evaluate Squeezer. The participants had between two and seventeen years of experience as game designers.

The evaluation was carried out in a two phase process. We recorded each video session and collected usage data (telemetry). The two test phases were designed so the participants would first test the interactive evolution interface, without being able to modify effects directly and later get full access. This way, they could provide focused feedback of that part of the interface. In the second phase, we allowed the participants to explore the effect manipulation more thoroughly. Here, we were gathering their feedback on the interplay between the interactive evolution interface and the effect manipulation. Our test design was inspired by tutorial design, where games often slowly increase the number and complexity of the shown features to allow the user to learn. The two test phases allow us to ask questions about the interface preferences among the participants, and the pros and cons of each interface.

During the test, participants were asked to open the software for the first time. Then they were introduced to the concept and purpose of Squeezer. They were guided to open the DISC ROOM demo scene, and add the interactive evolution scene, to begin the test. We asked them to evolve effects for the player character colliding with discs, death animations, as well as effects for disc bouncing, using the run-time interactive evolution interface. During the test they were encouraged to provide feedback on the process, the interface, and the generated effects while they worked. Once they had sufficiently explored interactive evolution and either felt satisfied or mentioned they wanted to manipulate or tweak effects more directly, the test proceeded to the second phase. In the second phase, they were asked to pause the evolution and have a look at the Unity inspector interface (see Fig. 3 and 1). From here they were instructed on how the interface worked, and the interaction between the inspector and run-time interactive evolution. They were allowed to mutate, regenerate and change anything manually, through the inspector interface, and they also had the option of resuming the evolution using the run-time interface. After about an hour, they were asked to provide immediate thoughts and feedback, to sent us their usage data and fill out a short questionnaire.

The survey asked them about their game design experience, and experience with adding effects to games. Additionally they were asked to provide feedback on interactive evolution as an introduction to Squeezer and effect generation, their thoughts on using interactive evolution as part of the workflow, and if they found it satisfying and/or surprising to work with. Lastly, they were asked if they would be interested in using a tool such as Squeezer in the future, or what would need to be changed in order for them to use it.

5 RESULTS

Given that we conducted expert interviews, we focused on individual statements and general agreements between the participants. Additionally to the expert evaluation of the tool, the user test also helped us to find bugs and usability issues. Since the user tests were carried out over a few weeks, a few bug fixes and quality-of-life improvements were added between the tests.

An example of this is the following: The first participant noted that, while working with interactive evolution, they did not want to change details of an effect manually. Rather, they wanted to change the effect slightly in a random directly. They suggested the addition of a mutate button for all effects in the effect tree, to allow for the mutation of a single effect or sub-tree. Because the underlying functionality was already in place, the button was added to the inspector interface before the next test.

Several users expressed at times that they wanted to roll back one generation and re-roll the current population. Some wanted to completely start over because they ended up with effects they didn't like. To avoid the latter and support the former, buttons to re-roll, restart and clear parts of the effects were added to the run-time interface between the second and third participant.

5.1 Qualitative User survey

The qualitative survey included questions about the user's proficiency in the area of game feel design. Four out of five participants write custom code or use libraries such as DOTween[14] for animations, and trigger the corresponding events themselves from code. But only two of the five were using any tools currently. One uses their own custom tools, the other one libraries like DOTween or Unity's Coroutines[15].

Three of the five participants found interactive evolution was beneficial to getting started with Squeezer and helped them get a perspective on the possibilities. One participant found it mostly useful as a way to discover new ideas, while the last participant found it "*cool*" but was missing some transparency in how the evolution actually would change the effects.

When asked about their thoughts on using interactive evolution in a tool like Squeezer, three of the participants said that they imagined it would be good for new designers and developers without the knowledge to "*easily make cool looking stuff*" or just as a tutorial for Squeezer. One participant found it "*chaotic, but also playful*" pushing them out of their comfort zone, while helping them to "*explore a possibility space too large to contain*" in their head.

Four of the participants found that the generated and evolved effects surprised them, and led to unexpected but interesting results. A few of the participants ran into bugs, that caused the exploration to mostly generate certain kinds of effects, and noted that it felt like the system mostly liked the color white and transparency effects.

Three of the five participants said they might use this in the future for game jams and personal projects, and a fourth said it might be useful for coming up with new ideas for effects. The last participant relies more on triggering effects manually from code, and suggested a more "lightweight" version of the API, would allow them to bypass Squeezers trigger system, but agreed that Squeezer should still have that capability as a visual design interface. One of the participants did raise the concern that Squeezer would probably not transfer well from the prototyping phase into production.

[14]http://dotween.demigiant.com/
[15]https://docs.unity3d.com/Manual/Coroutines.html

5.2 Participant Usage

The authors noted that the depth of the evolution was at most around ten steps. It usually took only two to four steps before users felt either content or wanted to start over. This could either be caused by the loose definition of the task they were asked to perform or by the fact that users felt they had run into “dead ends”.

The sound effect synthesizer caused several participants to mute the audio, because it kept generating oddly loud sounds. Squeezer allows full control and visibility of all the mutated values, providing users with insight into the changes the system is making [37]. One participant noted that the inspector interface has quite a lot of information. They proposed that adding icons and changing the names of effects could improve the user interface.

Several participants were surprised by the effects generated during evolution, and mentioned how those effects changed the game mechanics in surprising ways. Mixed-initiative systems can provide surprisingly creative insights when added to game artifacts [35]. To one participant Squeezer felt like “*an animation showreel for effects.*” and we took that as a compliment.

6 REFLECTION

In this paper we presented how adding interactive evolution to Squeezer can assist game designers in exploring game feel design [24] in game prototypes. The two interfaces for interactive evolution in Squeezer each have merits of their own. They can be functionally combined. However, the user experience of working with it, is still far from optimal. Future research is needed to explore ways of presenting a range of different effect variations for interactive systems. Previously, we created a Breakout clone with an AI playing the game [18], to allow for designing and testing at the same time. However, adding artificial agents to any game requires a lot of effort, and is thus a bad fit for the prototyping phase. While it is currently possible to edit the effect sequences while the game is running, this feature is not fully useful without a preview window.

While the effect tree generator supports generating effect sequences for a lot of opportunities in a game, several additional cases were discovered while preparing for the user testing. Categories such as ‘Starting/Spawning’, with objects fading, sliding or scaling into view, as well as options to distinguish between 2D and 3D effects would be great additions. Adding these additional categories and options would enhance the potential of applying Squeezer in game development.

Similarly, creating variations of the underlying handcrafted trees would allow an even more diverse set of generated trees to be created. This would present more variation to designers when generating effects for a certain event. Additionally it might be interesting to align variations across different event types, by creating for instance “Explosion type A”, which worked well with “Movement trail A” or “Muzzle flash A”.

Like with every prototyping tool, a potential pitfall of Squeezer is that the designer can become too attached to the placeholder effects. That they are locked into a particular path of exploration and limited in their creativity. The value lies in quickly exploring feasibility and examples of juice, and then being able to “start over” using the lessons learned from the prototype. Squeezer does not contain every possible effect, and it might even be missing some critical effects for game types we overlooked. However, it provides a good starting point for exploring and implementing effects early in a game design process, when the final look and feel has yet to be determined.

As one participant pointed out in their evaluation, Squeezer could benefit from streamlining the API for triggering effects directly from code. While this is already possible, with the current *Squeezer.Trigger(...)* API, there are several ways to tie code and effect descriptions together. An API that easily allows programmers to trigger, cache, mutate and save effects based on categories and intensity from code would open the evolutionary aspects of Squeezer to programmers.

7 FUTURE WORK

Comparing this version of Squeezer to the initial version presented and tested in [18], a lot has changed. The interface has been overhauled completely, many of the initial interaction design-concerns have been addressed. The generator, interactive evolution, and the mutation and locking system have been added. It is hard to compare the two interfaces, because the user interface has been upgraded with many small quality of life features. Yet, while we believe the new version is a lot better than the initial version, further studies of designers using Squeezer for their own projects, would highlight more thoroughly the issues and qualities of Squeezer.

As we stated in [24], Automated Game Generation [8] could benefit from the addition of effects. However, it requires implementing measure that make sure that effects are not conveying misleading feedback. A way of analysing what message a player gleams from a given effect sequence, and which effects fit generated game mechanics must be provided to add fitting feedback to generated games. Similar to having categories align across different events in a game, verifying that all effects match the game-play of a generated game consistently, requires further research.

8 CONCLUSION

This paper shows how generative and evolutionary strategies can be applied in game design, specifically in the realm of juice and feedback. It demonstrates how Squeezer, a tool for rapidly designing juice effects, became a “synthesizer”, combining modular sequencing and presets to generate complex chains of effects. The tool can help exploring the effect design space and it can also support determining which kinds of juice effects enhance feedback of a game prototype [18]. In order to further support this kind of exploration and discovery of the design space of game feedback, we expanded Squeezer with an interface for generation, mutation and interactive evolution of effects.

To verify the quality of our implementation, we conducted a user study with game designers who had no prior experience with the software. The study indicated that Squeezer could be a great tool to help designers discover and explore the possibilities of juice effects and interesting game mechanics. Additionally our test indicates that interactive evolution could ease the process of learning, creating and exploring juice effects with Squeezer.

ACKNOWLEDGMENTS

We would like to thank the test participants for participating and providing their feedback about working with Squeezer.

REFERENCES

[1] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. 1998. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Philip Bontrager, Wending Lin, Julian Togelius, and Sebastian Risi. 2018. Deep Interactive Evolution. In *Computational Intelligence in Music, Sound, Art and Design*, Antonios Liapis, Juan Jesús Romero Cardalda, and Anikó Ekárt (Eds.). Vol. 10783. Springer International Publishing, Cham, 267–282. https://doi.org/10.1007/978-3-319-77583-8_18

[3] Mark Brown. 2015. Secrets of Game Feel and Juice. https://www.youtube.com/watch?v=216_5nu4aVQ. https://www.youtube.com/watch?v=216_5nu4aVQ Accessed: 2020-04-17T11:52:14Z.

[4] Mark Brown. 2019. Why Does Celeste Feel So Good to Play? | Game Maker's Toolkit. https://www.youtube.com/watch?v=yorTG9at90g. https://www.youtube.com/watch?v=yorTG9at90g Accessed: 2020-04-17T13:21:54Z.

[5] N. Burtnyk and M. Wein. 1971. Computer-Generated Key-Frame Animation. *Journal of the SMPTE* 80, 3 (March 1971), 149–153. https://doi.org/10.5594/J07698

[6] Kitty Calis, Jan Willem Nijman, Terri Vellmann, and Adam Drucker. 2020. Disc Room.

[7] Jeff Clune, Jason Yosinski, Eugene Doan, Nabeel Samad, Sijie Liu, and Hod Lipson. 2012. EndlessForms.Com - Design Objects with Evolution and 3D Print Them! http://endlessforms.com/. http://endlessforms.com/ Accessed: 2020-11-17T17:22:46Z.

[8] Michael Cook, Simon Colton, and Jeremy Gow. 2014. Automating Game Design In Three Dimensions. In *AISB Symposium on AI and Games.* AISB, 1–4. http://research.gold.ac.uk/id/eprint/17354/

[9] Frank Force. 2020. ZzSprite - Tiny Sprite Generator. https://killedbyapixel.github.io/ZzSprite/. https://killedbyapixel.github.io/ZzSprite/ Accessed: 2020-11-17T17:25:13Z.

[10] Renaud Forestié. 2019. How to Design with Feedback and Game Feel in Mind - Shake It 'til You Make It. https://www.youtube.com/watch?v=yCKI9T3sSv0. https://www.youtube.com/watch?v=yCKI9T3sSv0 Accessed: 2020-04-17.

[11] Tracy Fullerton. 2014. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games* (3rd ed.). A K Peters/CRC Press.

[12] Kyle Gray, Kyle Gabler, Shalin Shodhan, and Matt Kunic. 2005. How to Prototype a Game in Under 7 Days. https://www.gamasutra.com/view/feature/130848/how_to_prototype_a_game_in_under_7_.php. https://www.gamasutra.com/view/feature/130848/how_to_prototype_a_game_in_under_7_.php Accessed: 2019-10-27.

[13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing Communications and Applications* 9, 1, Article 1 (Feb. 2013), 22 pages. https://doi.org/10.1145/2422956.2422957

[14] Kieran Hicks, Patrick Dickinson, Juicy Holopainen, and Kathrin Gerling. 2018. Good Game Feel: An Empirically Grounded Framework for Juicy Design. In *Proceedings of the 2018 DiGRA International Conference: The Game Is the Message.* DiGRA, 17. http://www.digra.org/wp-content/uploads/digital-library/DIGRA_2018_Paper_35.pdf

[15] Amy K. Hoover, Paul A. Szerlip, Marie E. Norton, Trevor A. Brindle, Zachary Merritt, and K. Stanley. 2012. Generating a Complete Multipart Musical Composition from a Single Monophonic Melody with Functional Scaffolding. In *ICCC*, Vol. PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL CREATIVITY. ICCC, Proceedings of the Third International Conference on Computational Creativity, 111–119.

[16] Robin Hunicke. 2009. Loving Your Player With Juicy Feedback. http://2009.dconstruct.org/podcast/juicyfeedback

[17] Mads Johansen, Martin Pichlmair, and Sebastian Risi. 2019. Video Game Description Language Environment for Unity Machine Learning Agents. In *2019 IEEE Conference on Games (CoG)*, Vol. 2019 IEEE Conference on Games (CoG). IEEE, 1–8. https://doi.org/10.1109/CIG.2019.8848072

[18] Mads Johansen, Martin Pichlmair, and Sebastian Risi. 2020. Squeezer - A Tool for Designing Juicy Effects. In *Extended Abstracts of the 2020 Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '20).* Association for Computing Machinery, New York, NY, USA, 282–286. https://doi.org/10.1145/3383668.3419862

[19] Martin Jonasson and Petri Purho. 2012. Juice It or Lose It. https://www.youtube.com/watch?v=Fy0aCDmgnxg. https://www.youtube.com/watch?v=Fy0aCDmgnxg Accessed: 2019-02-27.

[20] Jesper Juul and Jason Scott Begy. 2016. Good Feedback for Bad Players? A Preliminary Study of 'Juicy' Interface Feedback. In *Proceedings of First Joint FDG/DiGRA Conference*, Vol. Proceedings of first joint FDG/DiGRA Conference. DiGRA, Dundee, 2. https://www.jesperjuul.net/text/juiciness.pdf

[21] Jan Willem Nijman. 2013. The Art of Screenshake. https://www.youtube.com/watch?v=AJdEqssNZ-U. https://www.youtube.com/watch?v=AJdEqssNZ-U Accessed: 2021-04-22.

[22] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M. Lucas, Adrien Couetoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. 2016. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 3 (Sept. 2016), 229–243. https://doi.org/10.1109/TCIAIG.2015.2402393

[23] Tomas 'DrPetter' Pettersson. 2007. SFXR. http://www.drpetter.se/project_sfxr.html. http://www.drpetter.se/project_sfxr.html Accessed: 2020-07-11.

[24] Martin Pichlmair and Mads Johansen. 2021. Designing Game Feel. A Survey. *IEEE Transactions on Games* IEEE Transactions on Games (Early Access), IEEE Transactions on Games (Early Access) (2021), 1–20. https://doi.org/10.1109/TG.2021.3072241

[25] William T. Reeves. 1981. Inbetweening for Computer Animation Utilizing Moving Point Constraints. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '81.* ACM Press, Dallas, Texas, United States, 263–269. https://doi.org/10.1145/800224.806814

[26] Tom Schaul. 2013. A Video Game Description Language for Model-Based or Interactive Learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG).* IEEE, Niagara Falls, ON, Canada, 1–8. https://doi.org/10.1109/CIG.2013.6633610

[27] Tom Schaul. 2014. An Extensible Description Language for Video Games. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (Dec. 2014), 325–331. https://doi.org/10.1109/TCIAIG.2014.2352795

[28] Jimmy Secretan, Nicholas Beato, David B. D Ambrosio, Adelein Rodriguez, Adam Campbell, and Kenneth O. Stanley. 2008. Picbreeder: Evolving Pictures Collaboratively Online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08).* Association for Computing Machinery, Florence, Italy, 1759–1768. https://doi.org/10.1145/1357054.1357328

[29] Noor Shaker, Julian Togelius, and Mark Nelson. 2016. *Procedural Content Generation in Games.* Springer, USA. http://www.springer.com/gp/book/9783319427140

[30] Jiesang Song. 2005. Improving the Combat &apos;Impact&apos; Of Action Games.

[31] Steve Swink. 2007. Game Feel: The Secret Ingredient. https://www.gamasutra.com/view/feature/130734/game_feel_the_secret_ingredient.php?print=1. https://www.gamasutra.com/view/feature/130734/game_feel_the_secret_ingredient.php?print=1 Accessed: 2020-04-17T11:52:32Z.

[32] Steve Swink. 2009. *Game Feel.* Morgan Kaufmann.

[33] H. Takagi. Sept./2001. Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation. *Proc. IEEE* 89, 9 (Sept./2001), 1275–1296. https://doi.org/10.1109/5.949485

[34] Frank Thomas and Ollie Johnston. 1981. *The Illusion of Life: Disney Animation.* Abbeville Press, New York.

[35] Georgios N Yannakakis, Antonios Liapis, and Constantine Alexopoulos. 2014. Mixed-Initiative Co-Creativity. In *Foundations of Digital Games 2014.* Society for the Advancement of the Science of Digital Games, Proceedings of the 9th International Conference on the Foundations of Digital Games, 8.

[36] Jinhong Zhang, Rasmus Taarnby, Antonios Liapis, and Sebastian Risi. 2015. DrawCompileEvolve: Sparking Interactive Evolutionary Art with Human Creations. In *Evolutionary and Biologically Inspired Music, Sound, Art and Design.* Springer, Cham, 261–273. https://doi.org/10.1007/978-3-319-16498-4_23

[37] Jichen Zhu, Antonios Liapis, Sebastian Risi, Rafael Bidarra, and G. Michael Youngblood. 2018. Explainable AI for Designers: A Human-Centered Perspective on Mixed-Initiative Co-Creation. In *2018 IEEE Conference on Computational Intelligence and Games (CIG).* IEEE, Maastricht, 1–8. https://doi.org/10.1109/CIG.2018.8490433