Sky Labyrinth

PCG as a Driver for Efficient Level Design, Improved Usability, and Better Performance

Deniz S. Ozkaynak*
Variance Creative Studios
Armonk, NY 10504
deniz@variancecs.com

ABSTRACT

Exhibits how we leveraged Procedurally Generated Content[1] to empower level designers, improve the end-user experience, and enable performance gains. Showcases internal tools built by Variance Creative Studios developers, to convey one method of accessing the power of PCG.

CCS CONCEPTS

•**Applied computing** → **Computer games;** *Media arts;*

KEYWORDS

PCG, Object Pooling, Autorunner, Runner, Unity3D, Tools, JSON, Level Design, Rapid Iterative Testing, Iteration

ACM Reference format:
Deniz S. Ozkaynak. 2017. Sky Labyrinth. In *Proceedings of Foundations of Digital Games Conference, Cape Cod, MA USA, August 2017 (FDG 2017),* 4 pages.
DOI: 10.475/123_4

1 INTRODUCTION

PCG has long been used in game development and other applications. However as of late it has been a high profile phrase, often used to market a game's supposedly endless amount of exciting replayable content that often becomes stale quickly. Or similarly, a phrase used by managers and executives to cut corners and trim budgets, calling for less artistic labor modeling, texturing, rigging and lighting hundreds or thousands of objects. From the very beginning of development we at VCS knew we need not worry about algorithmically generating a plethora of content. What we needed was the ability to create a game that was fun, unique, and scalable. To develop Sky Labyrinth, we chose Unity3D as our engine chiefly because of the team's extensive experience with it and additionally due to the significant community support behind extending the Editor to build great assets or tools.

*Managing Partner, Lead Programmer
[1]PCG hereafter

FDG 2017, Cape Cod, MA USA

DOI: 10.475/123_4

2 BACKGROUND

We began[2] by designing and prototyping over the course of 8 months - a small indie team working evenings and weekends. We continuously iterated on the implementation of our core game design: an autorunner[3] that allows the player to go in many directions, rather than left or right. Dubbed "Omni-directional Runner", while the implementation details changed frequently, we remained true to this simple high-level design. Upon arriving at a suitably fun and playable implementation, based around solving mazes, we began building out a simple tool to create content titled **MazeGen**. When fed a number of prefabs[4] (components of a maze) along with a number of optional parameters, MazeGen instantiates a playable maze. As we would later come to realize, this was the embryo that would eventually grow into our suite of developer tools.

2.1 Game Overview

Overview of the look and feel of the game itself, somewhat important context to understand the goal and benefit of our approach to PCG. Please skip ahead to Subsection 2.2 at your leisure.

Tag: *Collect StratoSpheres, escape the labyrinth, or be trapped in the sky forever!*

Features:

- *Vertical Progression* - after beating each level, players see an overhead view of the next maze as they fall toward it
- *Mid-progression Controls* - players can move mid-air, influencing where they land in the next level to avoid obstacles, grab powerups, or optimize their route to solve the maze quickly
- *Hand-Crafted Content* - 30+ mazes each with unique level design, puzzles, and challenges
- *Play on both* desktop and mobile devices

[2]on July 1st, 2015
[3]a la Temple Run, Subway Surfers
[4]A Unity asset type that allows you to store a GameObject object complete with components and properties. Acts as a template from which you can create instances that inherit any edits made.

- *No Level Loading Screen* - Every time the game is launched, players face just a single loading screen no matter how long the play session

2.2 Tools Overview

Much like the previous subsection, provides somewhat important context but please skip ahead to Section 3 at your leisure, or consider watching this 2.5 minute video overview instead.

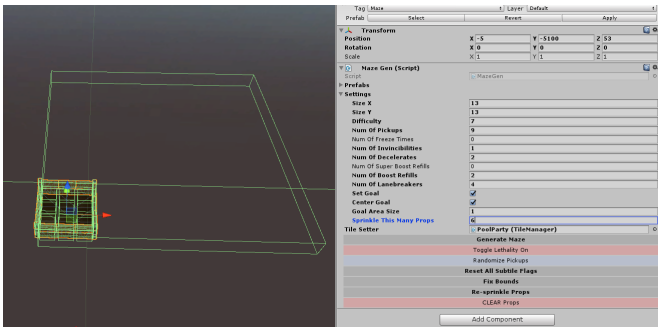

Figure 1: MazeGen.cs - before running
Accepts optional parameters such as dimensions, difficulty rating[5], the amount of each powerup to be randomly placed, where the level's Goal should be placed, and how many props to sprinkle around the maze.

Figure 2: MazeGen.cs - after instantiation
Don't like how this maze came out? Re-run the tool!

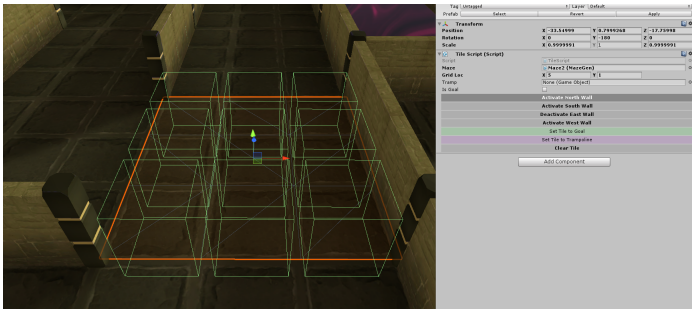

Figure 3: TileEditor.cs
Used to toggle walls on and off, move the Goal, or place puzzles.

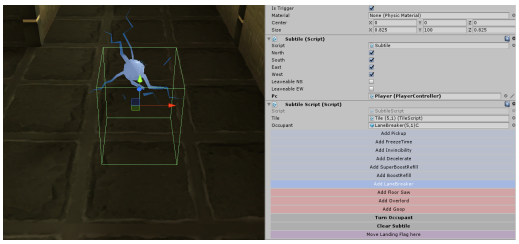

Figure 4: SubtileEditor.cs
Used to place individual pickups, powerup items, or obstacles. Also used to move the Editor Flag (see Figure 7).

Figure 5: WallEditor.cs
Used to place Wall Saw obstacles or break sections of the wall. Players can jump over broken wall sections, offering paths to avoid certain obstacles or shorten the route to success.

Figure 6: PillarEditor.cs
Used to cycle pillar models and graphics.

Figure 7: EditorFlag.cs - based on the configuration...
of the previous maze, this shows all possible cardinal directions the player could be facing upon landing. Very useful when designing levels, to prevent the frustration of landing and immediately running into a wall. Designers are able to move and rotate the Flag which automatically edits all previous mazes accordingly.

Figure 8: Pool Party Tools - when a designer is happy... with their maze changes, they use the "Write" tool to save all relevant data to JSON,[6] then use the "Delete" script to clear the Scene of all poolable objects. Nearly emptying the Scene enables faster load times[7]. When additional changes are needed, running the "Reinstantiate" script re-populates the Scene from JSON, ready to edit!

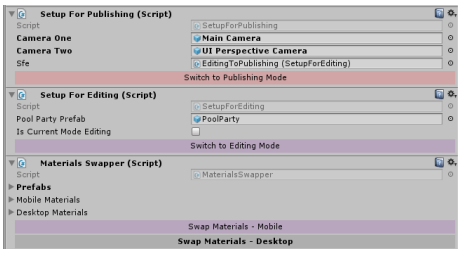

Figure 9: Publishing - many targetted mobile devices can't... handle the quality of assets, materials, particle effects, and SFX that desktop users can[8]. Certain camera effects are disabled, and nearly every single material has 2 versions, where the mobile version uses far more performant shaders. These tools automatically swap the assigned material for dozens of prefabs based on the target platform, making what would otherwise be a painful process relatively straightforward.

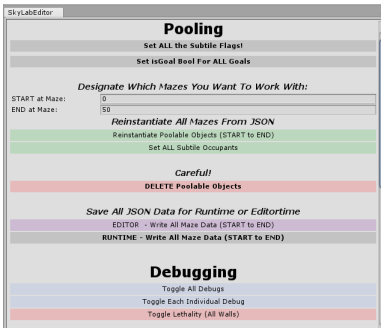

Figure 10: SkyLabEditor - a custom Editor window used... to interface with all these tools at once. Before we built this window, tools were scattered about attached to separate gameobjects or managers. As the suite of tools grew, collapsing everything into a single Editor window expedited level design and engineering tasks alike.

3 IMPACT

When it comes to enabling efficient level design, the previous section conveys how much time and effort can be saved with automated developer tools. Although we don't have any hard numbers, it's easy to imagine how many more man-hours it would take to build out mazes manually. We did do so a handful of times very early in prototype development; it's safe to say that the mediocre results of many painstaking hours was a partial motivator for developing our tools. Today, entire mazes can be designed on-the-fly and production ready in a couple hours or less, as seen in this timelapse of a design session where mazes 25 and 26 are built from scratch (and 27 is started) in 4 hours.

However the question remains: how do these tools impact end-users and runtime performance?

3.1 Improved Usability

More efficient design meant faster turnaround times on build releases. For the majority of our closed alpha and open beta testing periods we released builds on a weekly basis every Friday[9], gathering valuable feedback and feature requests. Over many months we received hundreds of replies about bugs, opinions, implementation ideas, and more! Unfortunately we did not have the resources to systematically approach usability testing. In lieu of asking testers to fill out surveys and trying to track statistics over time, we kept our ears low to the ground, so to speak. We interacted with the community often and nearly every interaction warranted a team discussion to some extent, leading to an assignment ticket being created, a feature being added, or a bug-fix expedited.

With time, Sky Labyrinth evolved from this buggy alpha[10] to a functional beta[11]. Disregarding the obvious changes in art direction, there's a noticeable difference in playability. If we look further down the road, here we see[12] a more mature version of the beta with many minor improvements. These refinements, such as smooth camera behavior[13] had been overlooked for a long time by our team[14]. One huge benefit of frequent release cycles was getting fresh eyes on our builds to identify overlooked areas for improvement. However the most impactful benefit actually came from our so-called power users; other game developers that took the time to write mountains of feedback on everything from minute details to huge implementation suggestions. For example /u/interestingsystems recently gave a feature suggestion that in retrospect we desperately needed:

> May I suggest that when the player turns the character into a wall and falls down, the game automatically realigns them back along a path when they get back to their feet. As a beginner it was frustrating to fall multiple times and then die as I was fumbling to get the hang of tapping at the right time to get back on track.

[9] for /r/gamedev's Feedback Friday thread on Reddit!
[10] Hyperlink to a GIF of an early alpha build
[11] A GIF of a more stable beta, not without its issues though
[12] A blog post showing a number of features or fixes via GIFs
[13] which came about only because certain users pointed out the jarring effect of a shifting field-of-view
[14] We had play-tested the game daily for so long, our eyes had become more than adjusted to little nuisances such as these

Failure had become a frustrating experience that we had approached with a number of different changes and tweaks, but this user's suggestion of an automated rotation after failure was far and away the most effective at eliminating frustration and getting users back to the fun. Although power-users like these are extremely appreciated and respected[15] there is one we have dubbed "The King of Feedback".

In March of 2016, /u/Saiodin went far above what anyone could ever hope or ask for. In addition to writing dozens of feedback paragraphs, this user created an animated GIF and later built an Unreal Engine 4 project to demonstrate a concept he was suggesting we implement. This directly altered our core movement system; although not fast or easy, refactoring our work led to far better performance and smoother gameplay.

Much further down the road, nearing today's version of the game, we continued receiving superb suggestions such as the addition of visual feedback[16] or this switch from a HUD element to a diegetic UI to represent one of the player's in-game resources.

3.2 Performance Gains

This section won't cover the general gains of implementing Object Pooling and PCG, as this design principle's pros and cons have already been well documented [1–4] in that regard. What we *will* cover is how *editor-time* tools can impact *runtime* performance, and to what degree.

3.3 Need More Tools

Early on in development, MazeGen was our only tool. However much like inheritance[17], over time many other programs were written based on MazeGen's design and implementation. Some tools were created to edit individual components of a maze and it's inhabitants, while others created to translate design work in-editor into poolable data. Other tools were created to assist with debugging, reconfiguring assets for certain platforms, and toggling features.

Each time we added prefabs to the list of poolable gameobjects, initial Scene loading time decreased slightly while disk usage increased by only a few hundred bytes. While a large object pool can have the downside of an initial 1 - 2 second spike in processing consumption during pool initialization, this is easily hidden behind our loading screen. At runtime performance hums, the construction and recycling of massive and complex mazes is unnoticeable to the player. Because we didn't need to consider the costs of creating new objects and collecting garbage, we were freed to focus engineering efforts elsewhere when time came for optimization and frame-rate centric development.

For mobile platforms, when more performant materials were swapped in the gains were huge, improving from barely playable on 2015 - 2016 devices to running well.

3.4 Disk Storage

As mentioned above, the more objects we decided to pool, the more disk space we needed to use on an end-user's device. Additionally certain objects, like obstacles with node-based pathing AI, need much more data saved than simply position, rotation, and scale. However even with a very complex, large maze, populated with plenty of entities, these storage files are under 50KB/maze on average and none over 75KB (yet!) The entire level design of over 30 mazes is stored on 1.5MB, a drop in the bucket for modern devices.

3.5 The Hidden Cost

With each new tool added, we coupled our systems further. In retrospect, much like inheritance, we ran into unnecessary coupling and an analogous version of the fragile base class problem. Although no tools actually inherited from MazeGen, its influence on their design and implementation acted like a base class. Because of this, and the tight coupling of components, making modifications to MazeGen or other important tools often led to unintended behavioral changes[18]. This should come as little surprise to the reader, but in retrospect designing the tools ecosystem from the ground up would have been effective and counteracting these problems. Rather than organically grow the suite of tools over time, based on needs of the team, a systematic approach to tools development would have saved countless man-hours. But alas, we had no idea we would need or develop all of these wonderful programs to aide our production when we began in 2015.

4 CONCLUSIONS

Admittedly the performance gains detailed are not earth shattering or contemporary; the benefits of PCG and pooling have long since been known and used. However, the real power comes from the duality of these tools. While making the task of designing and instantiating huge amounts of game content simple and efficient, *they also* provide well-known performance benefits. Although this suite of tools was not without its costs, if a team were to approach game development with the goal of building this type of ecosystem, suddenly certain costs can be greatly diminished. Easier said than done perhaps, something we very much intend to find out as we continue building games as Variance Creative Studios.

ACKNOWLEDGMENTS

The author would like to thank:

- Tyler Hiemke, for his contributions to MazeGen and other tools
- Ronald Mraz, for his game design expertise, incredible sound design, and 2 year dedication to the project.
- all of our brilliant alpha and beta testers

REFERENCES

[1] Jasper Flick. *Object Pools.* Catlike Coding. http://catlikecoding.com/unity/tutorials/object-pools/.

[2] Kevin Hoffman. 2005. *Pro ADO.NET with VB .NET 1.1.* Apress, Berkeley, California.

[3] Microsoft. 2017. *Improving Performance with Object Pooling.* Microsoft. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682822(v=vs.85).aspx.

[4] Peter Veentjer. 2010. *Java Extreme Performance: Part 2 fi Object pooling.* http://www.ctan.org/pkg/booktabs.

[15] each power appears in our in-game credits with a direct link to their feedback

[16] for when the player is slowed by enemies

[17] the design pattern, not the passing on of wealth

[18] bugs