HaptiEditor: Haptics Integrated Virtual Terrain Editing Tool

CÉLESTE MARUEJOL*, École de technologie supérieure, Canada
SEAN BOCIRNEA*, University of British Columbia, Vancouver, Canada
RISHAV BANERJEE*, University of British Columbia, Okanagan, Canada

The contemporary landscape of virtual world design is characterized by the ubiquity of diverse tools and terrain design engines, which have significantly reduced the barriers to entry in this domain. Despite this progress, the predominant input modalities of mice and keyboards fail to provide users with haptic feedback during the processes of designing, testing, and experiencing virtual environments. Addressing this limitation, we introduce HaptiEditor, a virtual terrain editing tool that integrates haptic feedback through the utilization of force-feedback capabilities offered by the Haply 2Diy device, implemented within the Unity game engine framework. Our primary objective is to enhance both the design process and the evaluative capacity of designers, as well as to enrich the immersive engagement of users or players navigating these virtual worlds.

Additional Key Words and Phrases: Haptics, Tool Design, Unity, Haply, ForceFeedback, Haptic Texture Rendering

1 INTRODUCTION

Haptics interfaces are often used for enhancing sculpting experiences. However, no significant implementation of force-feedback interfaces in the case of terrain editing and terrain painting has been created. Many existing haptic-augmented sculpting applications might be applicable to this task, but use haptic devices with three, or even six degrees of freedom, which are often both large and expensive. With HaptiEditor, we intend to use the target domain of terrain editing to our advantage, developing a compelling user interface with a relatively inexpensive two degree of freedom device. HaptiEditor is a project intending to explore this application of haptics, in the hope of evaluating the promise of our approach.

HaptiEditor is a Unity application which has for objective to edit maps and terrain by painting textures and objects on different scales. By using the Haply 2DIY we hope to create an interesting approach to terrain creation and edition through haptic feedback. The goal is to allow real time editing of a virtual space, and simultaneously feel the effect of the changes right away.

The development period span over a period 3 months during the Winter session of 2024, in the course entitled CanHap501. CanHap501 is a program which covers multiple Canadian universities in the hope of introducing graduate MSc students to haptic interfaces. This course teaches us how to conceptualize, prototype, develop and do user evaluation with multimodal human-computer interfaces and haptic experiences. This project is being developed in a team of three, each of us in different location (i.e. Montreal, Okanagan and Vancouver) with the management issues it entails. For instance, timezone issues as Okanagan and Vancouver are on a different timezone than Montreal, or versions of hardware given to each student were different depending on location.

Since the development timeline of this project is very short we decided to go with a rapid prototyping approach as learned in parallel during this course. Moreover, since we didn't have enough time to create a fully fledged terrain editor software, we agreed to use Unity to simplify a lot of the designing and implementation to get to experiment with

*All authors contributed equally to this project

Authors' addresses: Céleste Maruejol, École de technologie supérieure, 1100 R. Notre Dame O, Montréal, Canada, celeste.maruejol.1@ens.etsmtl.ca; Sean Bocirnea, University of British Columbia, Vancouver, 2329 West Mall, Vancouver, Canada, seanboc@student.ubc.ca; Rishav Banerjee, University of British Columbia, Okanagan, 3333 University Way, Kelowna, Canada, rishav.banerjee@ubc.ca.

the haptic side of the project quicker. Especially since Unity, as a game engine, already implements some solid systems for collision, forces and texturing.

HaptiEditor allows exploration of terrain at different scales via a novel zooming system. Using a system of sampling existing textures form the surface's material in order to create haptic feedback and Unity's collision system, HaptiEditor is able to provide different haptic feedback depending on the scale of the end effector scale in the scene. This allows a haptic continuum to enable the user to feel the terrain they are editing at any scale they would potentially need.

The present report is a statement of the advancements and findings made during the development of HaptiEditor. We will go over the different avenues tried in order to create HaptiEditor, what was prototyped and how it shaped the current software. We will then examine the results gathered through semi-formal user testing and the overall appreciation the project received. The results of the user testing and our implementation will thoroughly be discussed in the Discussions section. Appendix C goes over details of code judged the most important for our software to work.

2 RELATED WORK

Image-based Haptic Texture Rendering. Li et al. [3] produce a method for extracting normal forces from 2D images, which may then be rendered by a 3D haptic interface. The paper distinguishes between normal forces, acting in the vertical axis, and tangential forces, acting in the surface plane. In our work, we render these tangential forces and forego normal forces due to our use of a 2 DOF haptic device. We do not introduce a method for *creating* normal maps for our application, but instead reference the work of Li et al. [3] as an example of how one may extract normal information from image textures. Li et al. [3] display high rates of differentiability between represented textures with their method, though textures used in their evaluation are contrived and not based on real-world images.

Haptic Perception of Material Properties and Implications for Applications. Klatzky et al. [1] offer an overview of state-of-the-art approaches to haptic rendering of material properties. We bring attention to the discussion on texture representation, and note we use a normal mapping approach which allows for both direction and magnitude control of surface normals, forming a modified single-point probe model. Klatzky et al. [1] also note roughness and texture discrimination as a common evaluation metric for applications targeting textural rendering.

Hand Movements: A Window into Haptic Object Recognition. Lederman and Klatzky [2] catalogs exploratory techniques used when exploring physical objects. Our interface affords the patterns of lateral motion, static contact, pressure and contour folding, mediated through the single-point probe interface offered by our 2 DOF device. We note that these affordances were deemed by Lederman and Klatzky [2] to be sufficient (in that they allow performance better than chance) for texture and exact shape differentiation.

3 APPROACH

We attempt to deliver on a haptics focused terrain editing experience first and foremost. We first establish a reliable coupling via a virtual proxy to Unity's physics system, and then create a proof of concept terrain editing tool with force feedback driven by said proxy.

The three fundamental questions we had were as follows:

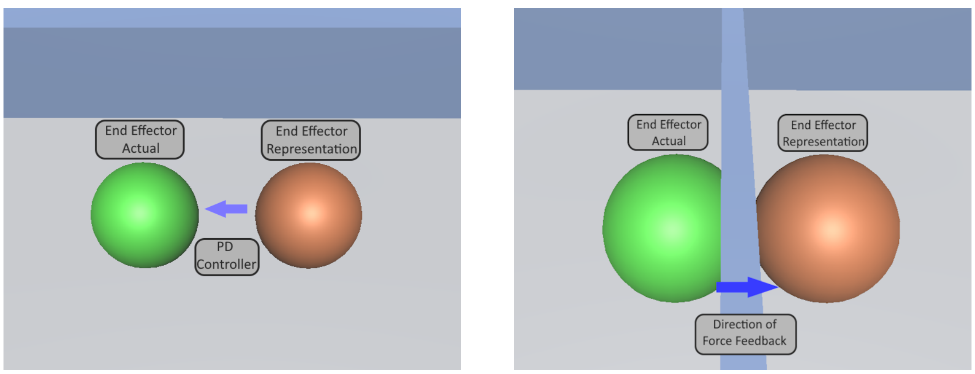
(1) How should we design a generic virtual coupling between Unity's physics engine and the Haply's force feedback mechanisms?
(2) How should we detect and render textures in real-time?

(3) How should we design the tool itself, with the main mode of interaction being through the Haply?

3.1 Designing a generic virtual coupling between the Unity physics engine and Haply device

We built off the Unity template obtained from the Haply GitLab repository as the foundation for our implementation. Upon closer examination, it became evident that the forces applied were hard-coded, prompting us to adopt a PD controller model [4] facilitated by a virtual proxy (*See 1*).

In our implementation, we utilize a Unity game object, "**End Effector Actual**" to track the ideal positional data of the Haply in the absence of obstacles in our terrain, proxied by another game object "**End Effector Representation**" which respects collisions with scene objects thanks to its built-in sphere collider, allowing it to interact with Unity's physics engine. Subsequently, we establish a PD controller relationship between these two entities. The underlying operational logic mandates the "**End Effector Representation**" to consistently attempt to minimize the euclidean distance between itself and the "**End Effector Actual**". This behavior is governed primarily by the proportional component of the PD controller, supplemented by the derivative component to offer additional smoothing (*See 1a*). In instances where the "**Representation**" detects any physical collisions, it directs the Haply to exert a force in the direction of the "**Representation**" from the "**Actual**" (*See 1b*). This happens in parallel to the distance minimization attempts of the "**Representation**". Consequently, this establishment facilitates an adaptable virtual coupling mechanism, subject to the influence of Unity's physics engine via the intermediary proxy.



(a) PD controller moves Representation to Actual if there is no obstruction

(b) Force Feedback Rendered if physics collision is detected

Fig. 1. Virtual Coupling between Representation and Actual End Effector. Note that the **"End Effector Actual"** is never visualized in the tool itself.

Notably, while the direction vector is three-dimensional, only the X and Z components of this vector are translated to the Haply to render force feedback. The selected axes are simply an artifact of our design decision for editing on a terrain lying in the X-Z plane.

3.2 Generating and rendering textures in real-time

Tangentially influenced by the research conducted by Li et al. (2010) [3], we opt to leverage the inherent image data embedded within the texture for the application of small directional jitters. Our initial approach by sampled a three by three pixel window beneath the end effector representation, subsequently extracting the brightness values of each pixel.

Each pixel then exerted a force on the end effector away from itself proportional to its brightness. This mechanism imparts the perceptual impression of being coerced towards regions of lower luminosity, which can intuitively be mapped to "lower points" in the texture.

However, our development environment affords us a different option. Unity, being a game engine, supports normal-mapping for textures, used in game development for more realistic rendering of surface features. Normal maps contain for each pixel a normal vector representing the direction of the surface of the material, allowing us to compute from a single pixel the direction in which a probe (our end effector) should be pushed by a surface interaction from a single pixel sampled from the normal map. We thus save both memory accesses and computation time, improving simulation speed. Normal maps for in-game materials not only commonly available, but are usually generated programmatically from a sculpted surface texture, and thus also offer improved accuracy without incurring significant overhead to potential users.

Critically, texture-driven force modulation only manifests during end effector movement, thereby avoiding any undesirable tremors during static user positioning. By recalculating this force on a per-frame basis, an appreciable frequency modulation is introduced to the end effector, directly mirroring the texture's visual attributes (*See 2*).

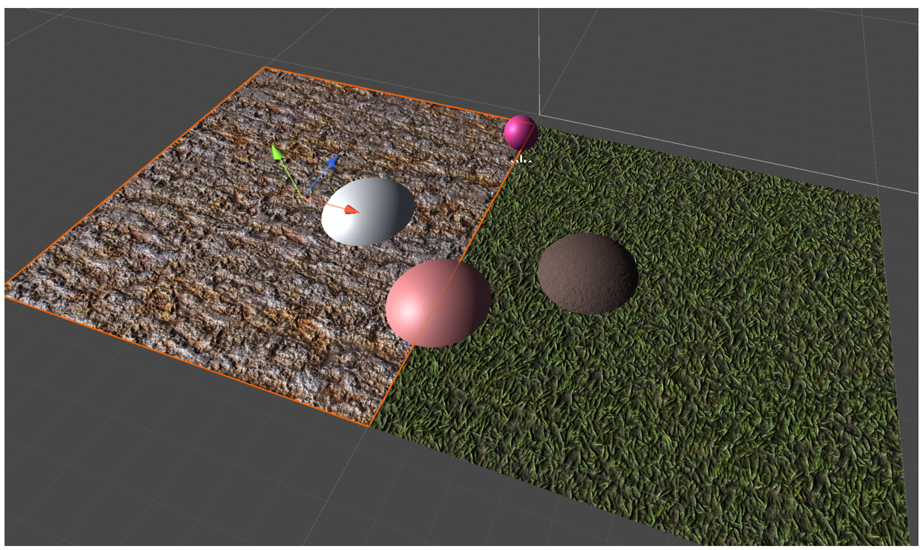


Fig. 2. Two different textures providing different jittery sensations

3.3 Tool design and Haply interaction

Game engines typically offer a range of tools for terrain editing, which are primarily oriented towards editing within the editor environment rather than functioning during runtime. Consequently, it became necessary for us to reconstruct the fundamental components of a terrain editing tool within Unity, essentially embedding them as "game mechanics". This endeavor primarily involved leveraging Unity's Terrain game object [5], which is specially built to optimally encode localized texture and static object placement data. A straightforward user interface facilitates the selection among textures, objects, and an eraser tool, further categorized into sub-menus delineating texture types (e.g., grass, sand) and object variants (e.g., trees, rocks) (refer to Figure 3). The user can then designate their preferred object or texture through mouse interaction. Upon selection, the Haply device acts akin to a virtual paintbrush, allowing object placement or texture painting at the location of the end effector. Activation of the painting action is achieved either through the stylus button integrated with Gen3 DIY devices or, alternatively, by utilizing the space-bar in cases where the stylus button is unavailable.

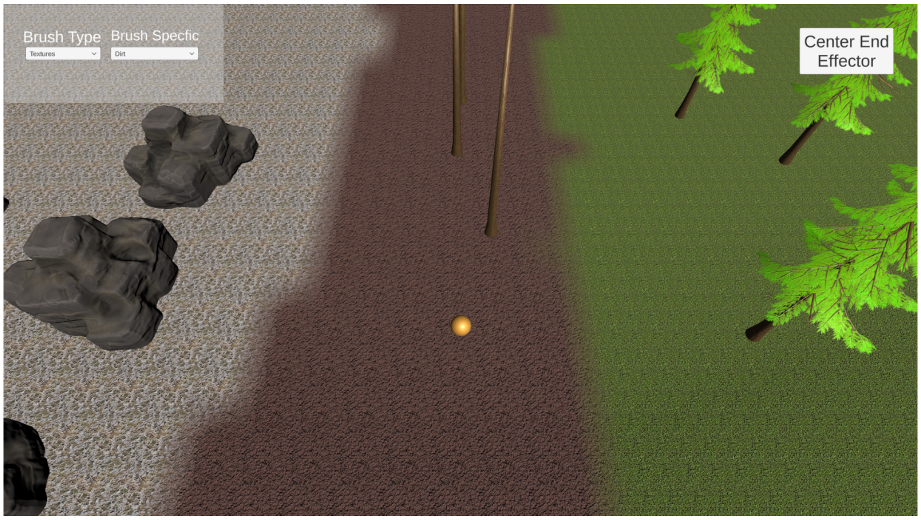


Fig. 3. The final terrain painting interface

The central utility of this approach lies in its capacity to provide users with tactile feedback with their interactions with the terrain in real-time. Objects impart a rigid collider-based resistance as a consequence of the virtual coupling (see Subsection 3.1), while textures provide haptic modulation based on their visual characteristics and behavior in accordance with haptic texture rendering principles (see Subsection 3.2).

4 PROTOTYPING

4.1 Developing for Unity

There were two main aspects to developing for Unity which we rehaul from the sample Unity Haply Gitlab repository:

4.1.1 Board Configurations. Since we have differently configured 2DIY Gen 3 Haply boards (and in some cases, Gen 2 boards), we need a way to switch easily between these configurations without spending time changing code whenever we push code to each other. Given that all the team members in this project worked remotely and in different time zones, this was imperative to a smoother development experience.

To facilitate this, we change the flow of logic for initialising the board to use configuration files, refered to as scriptable objects in Unity. By storing our different configurations in these files, we can easily swap in the appropriate configuration during development time without affecting any of the actual code.

4.1.2 Utilising Unity's Physics. The main benefit to utilisng Unity was to leverage it's physics engine for automated haptic experiences. The process of connecting the Haply to Unity physics has been described in detail in 3.1. There are minor nuances in the implementation itself, specific to Unity. These include making sure the physics engine is running

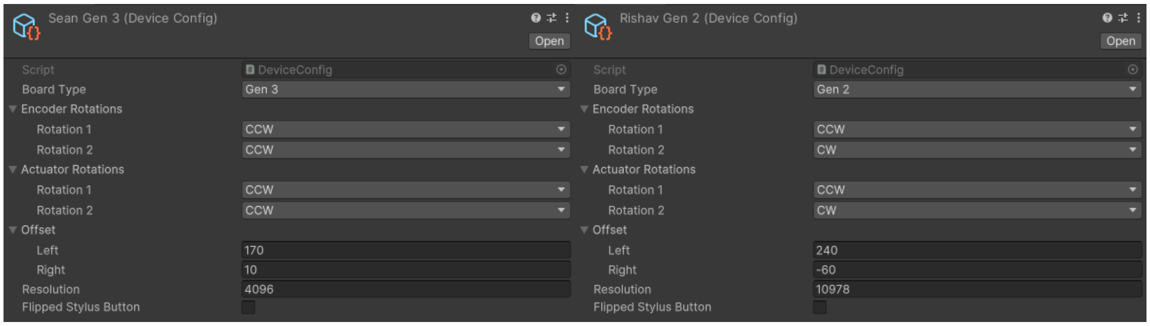


Fig. 4. Different board configurations for the Haply

at a higher framerate than normal since the physics is separated from the graphics, enabling continuous collisions for a smoother experience, and

4.2 Zooming

4.3 Painting

The incorporation of painting capabilities for objects and textures within a terrain editor is imperative for a multitude of reasons. Firstly, such functionality facilitates the creation of intricate and visually captivating landscapes by allowing users to precisely apply diverse textures and objects onto terrain surfaces, thereby enhancing realism and aesthetic appeal. More specifically, with the ability to add their own textures and objects, this feature enables users to customize their environments with precision, enabling the realization their artistic visions.

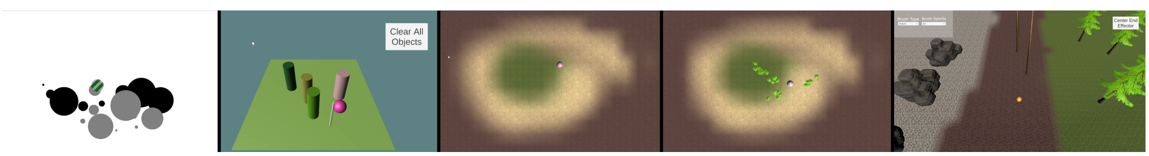


Fig. 5. Representation of the evolution of the prototypes throughout the development of the project (1) shows screenshot of the first prototype made to validate our first iteration (2) reprensents a screenshot of the first prototype after some code clean up and transfered to 3D space (3) displays a screenshot of the texture painting working on Unity's terrain game object (4) illustrates painting objects on the the terrain previously textured (5) is a capture of the final version which can be seen in Fig. 3

In line with the comprehensive nature of this project, the process of painting underwent multiple iterations of prototypes throughout the development phase. In the forthcoming discussion, we shall examine each prototype and elucidate the insights gleaned from them. The creation of our initial prototype, referred to as "the sprinkler," transpired towards the end of the first iteration. Its primary objective is to validate the efficacy of the PD controller implementation and the integration of Haply's API for the third iteration of Haply's 2DIY. This prototype functioned as a testament to the feasibility of dynamically generating objects during runtime and eliciting force-feedback from interactions between the end-effector and these aforementioned objects. As illustrated in Figure 2. (1), the sprinkler is placing gray circles as long as we were pressing the stylus button or the spacebar[1]. These gray circles serve as visual indicators devoid of

[1]the spacebar is used as backup throughout the project if the stylus button doesn't work for diverse reasons

collision detection, providing a prelude to the subsequent placement of black circles, complete with colliders, upon the eventual release of the stylus button or the spacebar key.

Just after finishing the transition from 2D to 3D, we repurposed the code of "the sprinkler" to be usable in 3D space since it is a quick way to, once again, determine wether or not everything is working as intended. We are reusing the same logic behind the placement of object as it has proven effective for the first prototype. In the second image of Figure 2., we can see that the scene is now in 3D and what was previously circles are now cylinders with random colors. The force-feedback is kept the same, if we move the End-Effector into a cylinder, we will be pushed out as if it is a wall. We concluded from this prototype that handling the object placement using a similar script is aligned with our goals for object placement, especially since it is easy to use and learn how to use it, thanks to the stylus affordance.

The subsequent prototype aims to enhance information retention across multiple executions. Our approach pivots towards leveraging Unity's terrain game object, which inherently retains terrain deformation, applied textures, and placed objects. This strategic alignment enables seamless integration with Unity's terrain editing system, obviating the need for extensive bespoke implementation. This not only yields temporal efficiencies but also enhances usability, capitalizing on user familiarity with the platform. Our primary focus lies on object and texture painting, thereby excluding terrain elevation adjustments.

In the prototype corresponding to Figure 2. (3), we introduced the capability to paint textures via mouse input. Employing mouse-based prototyping expedites debugging processes, preempting potential issues arising from haptic interfaces. This prototype proved its importance in facilitating rapid parameter experimentation, refining aspects such as brush radius, fall-off characteristics, and curve adjustments to ensure smoother edge rendering during texture painting. From this prototype as a base, we implemented object creation for terrain and object deletion, still utilizing the mouse as an input. Similarly, it permitted us experiment with different parameters to find what feels best, user experience-wise. The results are displayed in Figure 2. (4).

Subsequently, the amalgamation of prototypes culminated in a singular definitive outcome, as depicted in Figure 2. (5). Firstly, we transitioned the painting process to be contingent upon the positional data of the end-effector representation. Secondly, we repurposed the coroutine mechanism utilized for object painting from the second prototype, integrating it seamlessly with newly devised functionalities for object and texture painting, as well as object erasure. Lastly, we devised a concise user interface, affording runtime modifications of tools and their respective painting attributes.

4.4 Texture

5 EVALUATION AND RESULTS

Evaluation took the form of a user study, in which we provided a directed walkthrough of our work to novice users, after which we asked a nine linear scale questions about their experience. We walked users through as follows:

(1) The user was briefed on the purpose of the project, and that the end effector would be be their point of interaction with the terrain, both for painting and for feedback.
(2) The user was directed through the menu and setup screens.
(3) The user was shown the texture selection menu and told to paint different textures.
(4) The user was allowed to explore this functionality to their satisfaction.
(5) The user was shown the object selection menu and told to paint different objects.
(6) The user was allowed to explore this functionality to their satisfaction.
(7) The user was shown the object deletion menu and told to delete existing objects.

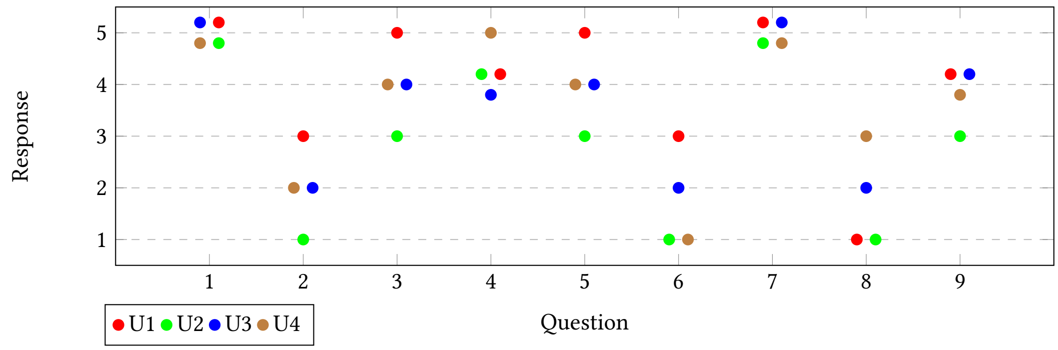


Fig. 6. User Study Responses

(8) The user was allowed to explore this functionality to their satisfaction.

After this experience, the user was asked the following set of nine 5-point linear scale questions, and told to answer between 5 meaning "most" and 1 meaning "least":

(1) How easy was it to tell the difference between feedback from objects and feedback from surface textures?
(2) How easy was it to tell the difference between feedback from different surface textures?
(3) How easy was it to tell the difference between feedback from different objects?
(4) How easy was it to tell the difference between feedback from all sources at different zoom levels?

(5) How helpful was feedback from objects in understanding and navigating the current state of the terrain?
(6) How helpful was feedback from texture in understanding and navigating the current state of the terrain?

(7) How synchronized was haptic feedback with the terrain you saw on screen?
(8) How physically fatiguing did you find the haptic feedback?
(9) How would you rate your overall enjoyment of the experience?

The responses to these questions is charted in Figure 6; we provide data on four respondents, grouped by color. Overall sentiment was positive with respect to force feedback, with all respondents rating both the utility and differentiability of shape-driven feedback highly. Users generally found it difficult to differentiate between textures, and did not find them helpful for terrain navigation. We conjecture improving texture differentiability would improve the usefulness of this feedback, though in its current state textural feedback still provides users feedback on scale and rate of motion. Users did however find it easy to discriminate scale on the basis of haptic feedback. Users generally did not find the experience fatiguing, and found it both well-synchronized with the visual terrain rendering and enjoyable overall.

We conclude that users generally felt that feedback from scene objects was both salient and useful, but that textures fell short due to difficulty in differentiation. However, the positive responses for both sentiment, fatigue, and usefulness of shape-driven feedback supports the viability of the project.

6 DISCUSSIONS AND LIMITATIONS

6.1 Discussions

The Haptic-Editing process. Through our evaluation, we discovered that users generally appreciated shape-driven haptic feedback, as it is salient, easily discriminated, offers practical uses like density estimation, and prevents users from overpopulating terrain regions. We also learned that texture feedback as we implement it in this work is of limited usefulness, as many users complained of poor differentiability. Use of a 2D pantograph interface to populate a 2D terrain object was grasped intuitively, and none of our participants had difficulty understanding the connection between the physical end effector and the representation of the end effector in our editor. We claim our interface would be well-suited as a plugin to Unity, allowing game developers to continue using tools they're familiar with, but with the added benefit of haptic feedback for terrain editing and similar applications. Indeed, force and texture feedback are both driven by parts of the engine, colliders and normal maps respectively, that will necessarily be used by a game developer in this context, and thus our interface requires no additional burden to use.

Unity as a tool for haptics design. Throughout history, various art forms, such as music, drawing, photography, film, video editing, game development, and XR development, have experienced significant advancements whenever developers have embraced user-friendly tools. These advancements have been facilitated by the availability of accessible cameras, freely available software for music composition and video editing, as well as widely supported game engines like Unity and Unreal. However, when the primary means of entry into these fields is limited to Java and Processing code, the potential for designing experiences becomes constrained by the technical skills of developers, creating a bottleneck effect. It is imperative to encourage the haptics community to explore beyond mere code frameworks and instead adopt a singular, user-friendly engine (such as Unity) that can empower designers to focus on the user experience aspect of haptics, rather than being bogged down by technical jargon. Such an approach would enable individuals to specialize in various aspects of haptics, be it hardware, software, or design, akin to the specialization seen in the game development industry, thereby enhancing the overall quality of output.

6.2 Limitations

Movement in an infinite space. With our current implementation, the end effector can only move in the virtual space a distance proportional to the Haply device's arm length. While we can change the movement scale in the engine, the end effector will eventually get stuck due to the haply's arm pantograph constraint. The primary difficulty lies in changing the relative positioning of the proxy with respect to the world, and what the underlying user experience design philosophy should be for the same without disorienting the user.

Fine grain control of placed objects. While we have the ability to place objects around a specific space, we do not have the ability to move or rotate a placed object in any degree of freedom, or scale the object up and down. This was an initial consideration our project had but had to be discarded in the interest of time and producing a working prototype. The main issue with this comes in tackling the user interaction model to edit the transform data of an object. Since the haply is the main mode of interaction, we could consider using it similar to a mouse, and designing movement, rotation and scale gizmos that can subsequently be used for the editing process.

7 ACKNOWLEDGMENTS

We would like to thank all the professors of the CH501 course for providing us this unique opportunity of learning and developing haptic experiences from scratch. This includes Dr. Oliver Schneider, Dr. Karon MacLean, Dr. Jeremy Cooperstock, Dr. Pascal Fortin, Dr. Vincent Levesque and Dr. Pourang Irani. We would also like to thank Dr. Antoine Weill-Duflos from the R&D department of Haply Inc. for providing technical assistance during the developing for the Haply. Finally we would like to thank the Teaching Assistants Bereket Guta, Anuradha Herath, Sabrina Knappe and Juliette Regimbal for guiding us through the various challenges in the course and our project.

8 CONCLUSION

Throughout the duration of this project, our conceptualization has undergone iterative refinement, integrating insights gleaned from collaborative discussions and the outcomes of prototyping endeavors. Initially, our project envisioned the development of a comprehensive terrain editor operating within three-dimensional space. However, subsequent deliberations with our instructors and internal team discussions led to the realization that the Haply 2DIY platform lacked the requisite capacity to accurately perceive depth within a three-dimensional environment. Consequently, we resolved to focus primarily on surface-level terrain features, directing our attention towards the painting of objects and textures while retaining the innovative capability to manipulate scale.

As a result of these findings, we concluded that the most judicious course of action entailed positioning our project as a complementary plugin within the Unity ecosystem, specifically tailored to augment the functionality of Unity's terrain game object. In essence, our endeavor reframes the editor as an extension of Unity's game engine, enhancing the user experience by facilitating the intuitive painting of objects and textures via the Haply 2DIY.

Amidst our prototyping endeavors, we encountered the unexpected ease of transitioning from haptic texture to haptic force feedback. Leveraging Unity's foundational concepts of textures and colliders, we observed that altering the scale of the End-Effector representation sphere significantly influenced its collision behavior with surrounding objects. At certain scales, the End-Effector exhibited the ability to navigate on top of obstacles that previously blocked its progress.

Despite the current rudimentary state of our project, we believe it effectively showcases the future possibilities granted by such integration. Especially providing the encouraging results we received from our user evaluation and testing. Moreover, the extensibility of the current codebase utilizing Unity's robust scripting systems, which lay a solid foundation for future expansion. For instance, there are multiple ways to bring enhancements to the projects, one of which would come in form of improvements and additional functionalities to the painting UI such as being able to change the brush radius. Additionnaly, experimentation is required to counteract our current limitations and reinforce our texture haptic feedback.

REFERENCES

[1] Roberta L. Klatzky, Dianne Pawluk, and Angelika Peer. 2013. Haptic Perception of Material Properties and Implications for Applications. *Proc. IEEE* 101, 9 (2013), 2081–2092. https://doi.org/10.1109/JPROC.2013.2248691

[2] Susan J Lederman and Roberta L Klatzky. 1987. Hand movements: A window into haptic object recognition. *Cognitive Psychology* 19, 3 (1987), 342–368. https://doi.org/10.1016/0010-0285(87)90008-9

[3] Jialu Li, Aiguo Song, and Xiaorui Zhang. 2010. Image-based haptic texture rendering. In *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry* (Seoul, South Korea) *(VRCAI '10)*. Association for Computing Machinery, New York, NY, USA, 237–242. https://doi.org/10.1145/1900179.1900230

[4] MathWorks. 2011. What is PID Control? — mathworks.com. https://www.mathworks.com/discovery/pid-control.html. [Accessed 18-04-2024].

[5] Unity Technologies. 2014. Unity - Manual: Terrain — docs.unity3d.com. https://docs.unity3d.com/Manual/script-Terrain.html. [Accessed 18-04-2024].

A VIDEO FIGURE

Please find a 2-minute overview of the functionality of our project here:
https://www.youtube.com/watch?v=PPx2JuhQ9Us

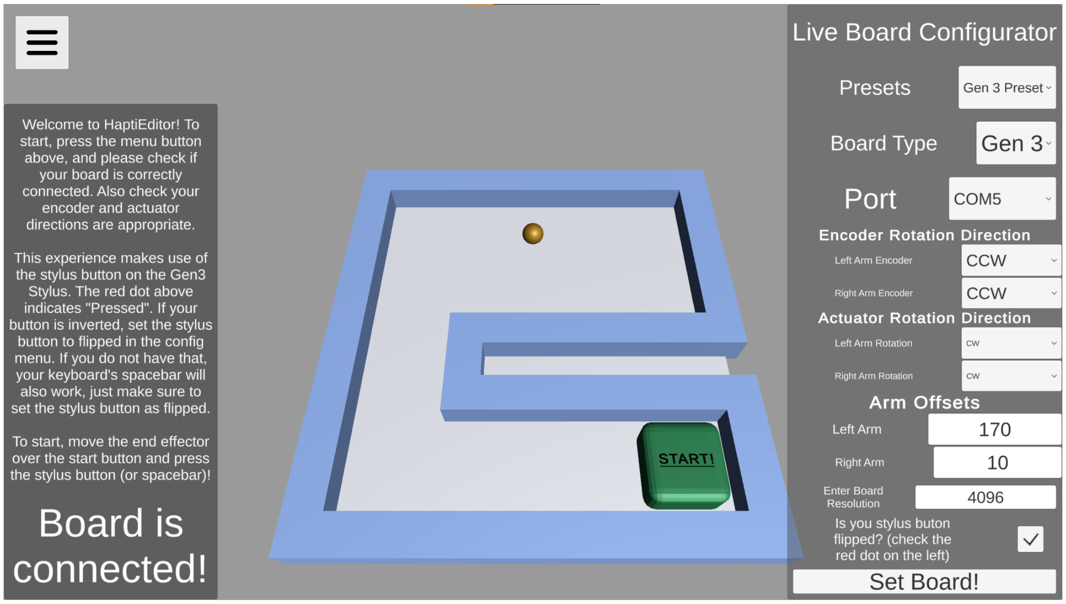


Fig. 7. HaptiEditor configuration menu

B INDIVIDUAL CONTRIBUTIONS

B.1 Rishav's Contributions

For the final iteration, we wanted to build an executable that anyone with a haply 2diy board can plug and play. Expecting everyone to install Unity would be quite an ask, so we needed to build a live board configurator.

The live board configurator would need to modify the following parameters:

- Board Presets
- Gen2 or Gen3 (for arm distance offset)
- Encoder rotation direction
- Actuator rotation direction
- Arm offsets for base position
- Encoder resolution
- Flipped Stylus Button (Some Gen3 boards have flipped sensor data for the stylus port)

Actually passing the appropriate data and reloading the board was significantly more difficult. In a nutshell, I had to do the following:

- Cancel the worker thread simulation task gracefully
- Flush all forces
- Delete the existing instance of the board (along with the encoder, actuator and sensor parameters)
- Create a new board instance with the new parameters
- Attempt a connection to the new specified port based on the user's selection from current active ports
- Launch a new worker thread.

- Potentially connect to a Button Handler if the scene had one present.

This required a significant amount of refactoring of the core hAPI, but in the end I was successfully able to load and reload new user specified board configurations. The main chunk of this was happening in the `EndEffectorManager.cs` as follows:

```
public void ReloadBoard(DeviceConfig customConfig, string targetPort)
{
    // Destroying existing setup
    haplyBoard.DestroyBoard(); // added function to close port and destroy this board
    CancelSimulation();
    SetForces(0f, 0f);
    Destroy(pantograph.gameObject.GetComponent<Device>());
    // New Setup
    device = pantograph.gameObject.AddComponent<Device>();
    device.Init(); // re-establishes basic connections with pantograph and board
    LoadBoard(customConfig, targetPort);
    // Checking for button handler
    ButtonHandler buttonHandler = gameObject.GetComponent<ButtonHandler>();
    if (buttonHandler == null) return;
    buttonHandler.SetButtonState(customConfig.FlippedStylusButton);
}

private void LoadBoard(DeviceConfig customConfig = null, string targetPort = null)
{
    device.LoadConfig(customConfig);   // loads new config if custom config is not null
    haplyBoard.Initialize(targetPort); // attempts connection with new port
    device.DeviceSetParameters();
    // ...
    simulationLoopTask = new Task( SimulationLoop );
    simulationLoopTask.Start();
}
```

After this, I decided to improve the visual experience of the project. Users will be expected to understand that they have to configure their board, and that their board might be set up different to our dev setups, so the menu scene should ideally be different from the actual terrain editing scene. Additionally, the stylus button (or space bar) is critical to the experience, so the users should be aware of how to do that as well.

To let the user learn this separately, I worked on a simple menu scene with a bit of flair, and added scene transitions. Users will now be expected to first configure their board, be able to move over a physical button in the world, and then click on the stylus button to enter the terrain painter tool.

I fished out a basic scene manager and transition handler from an older project, and after some tweaking, blender modelling and building an executable for Windows, I produced the menu scene in Figure 7.

B.2 Sean's Contributions

This iteration, I finalized work on the texture pipeline and integrated it with Pierre's work from iteration 2 on texture and object painting on the terrain object. Pierre wrote some logic to get world position into a corresponding position on the surface of the terrain object, removing the need for our expensive physics ray-cast. The rest of the process involved the following changes:

- Detecting which terrain texture was currently painted onto the surface of the terrain underneath the end effector. This required fetching the blend ratios of each material at the EE location and determining which was present:

```
float[,,] swatch = terrain.terrainData.GetAlphamaps((int)pixelUV.x, (int)pixelUV.y, 1,1);
```

- Once I have the ID of the texture to sample, we fetch color from its normal texture, giving us both a direction and intensity of force we immediately apply to our end effector. This greatly simplified the sampling code:

```
Color normalPixel = prot.normalMap.GetPixel((int)normalUV.x, (int)normalUV.y);

forces.x += normalPixel.r - 0.5f
forces.y += normalPixel.g - 0.5f;

forces *= intensity;
previousPosition = eeTransform.position;
```

- I introduced a variable `swatchscale` to allow us to control the ratio of world movement relative to movement on the normal map, controlling the linear density of our texture. Normal sampling also greatly improved the accuracy of forces, as we're no longer estimating heightmaps from surface color but now have geometry-accurate normal maps.
- Selected texturally distinct normal maps for our materials for a clearer distinction between painted materials.
- Added back space-bar support for painting, so a user has an alternative to the stylus button.

I then tuned the scaling code Rishav worked on in iteration 2 and added it to our scene. The following changed:

- I added a scale factor for the movement range of the end effector, so it expanded as the scene zoomed out, covering the new area.
- I then tuned the ratios for movement range, end effector size and camera zooming so that they scaled in tandem.
- I changed the painter code so that the painted objects had their appropriate colliders, allowing the large end effector to skate over rocks as we'd intended it to, rather than getting stuck like before.

Lastly, I tuned the texture sampling code again so that it would play nice with the zooming feature we'd introduced. I chose a `swatchscale` such that at high zoom levels, individual surface features like pebbles were quite large and discernible, but with a wider camera, surface features became higher frequency noise and force feedback from geometry became the primary force on the end effector. Textures representations were different depending on the painted texture on the terrain and could be rendered in tandem with terrain objects placed during use. We learned that these pipelines could all coexist in a cohesive way and were pleased to see our parallel approach to developing them paid off.

Post iteration 3, I fixed some outstanding bugs, worked on my section of the report, and prepared the video figure.

B.3 Pierre's Contributions

As stated in our blog posts for iteration 2, the goal for the third was to merge every concepts and prototypes into one single, preferably enjoyable, experience. In this iteration we ended up working together way more and in closer collaboration by helping each others a lot more. This can be easily explained because we didn't prototype on a different aspect of the experience and were actually making something unique together. This is why sometimes the lines of contributions of what I did and what a team member did will blur into what we did this feature together.

While we knew that we would merge all of our progress into the Terrain scene because the end goal is to use Unity's Terrain game object has our editor, the `TerrainScript.cs` from iteration 2 wasn't usable as is. Firstly, we need to fix the lack optimizations. Secondly, the user should have a way to change the brush type and their specifics without using

Unity editor menus (it is necessary if the user interacts with our software through an executable instead of the Unity Project). Thirdly, Lastly, it is unlikely the code we used for texture sampling for haptic feedback would work on the Terrain game object, because it has the particularity to not have a `MeshRenderer` (a component that allows a game object to render a mesh). Finally, there needs to be a way paint using the stylus button instead of the mouse.

During this iteration Rishav did an amazing work at going over the code that has been made and telling us what should be avoided in the future. Following those practices we started a refactoring on `TerrainScript.cs`. During this time, I found a way to optimize the management of the `TreeInstances` what were giving us issues during the second iteration, basically, deleted `TreeInstances` would never really be deleted but replaced with empty version of themselves.

This is problematic because with the core logic of the problem they would be given another collider the next time the software is running even though there is nothing to delete.

```
private void OnApplicationQuit()
{
    //test
    List<TreeInstance> trees = new List<TreeInstance>(terrain.terrainData.treeInstances);
    List<TreeInstance> trees_cleaned = new List<TreeInstance>();
    TreeInstance empty_tree = new TreeInstance();
    for (int i = 0; i < trees.Count; i++)
    {
        if (!trees[i].Equals(empty_tree))
            trees_cleaned.Add(trees[i]);
    }
    terrain.terrainData.SetTreeInstances(trees_cleaned.ToArray(), true);
}
```

Using this code when the application is closed we remove all of the `TreeInstances` that we could consider empty. The way it works is by going through all the objects in the Terrain `TreeInstances` and comparing it with an empty object. If they are different we add them to the new list that will contain all the non-empty `TreeInstance`.

Then using the `ObjectPlacer.cs` as a reference I created two co-routines one for painting object on the terrain and the other for painting texture. We implemented an enumerator containing all the brushes types (i.e. Texture, Object and Object Eraser) and depending on this brush type one of the co-routine or the object deletion will be enabled.

From then on Rishav worked with me on a basic UI so we can change the brush types and which texture/object is being painted.

C CODE