# GoThrough: a Tool for Creating and Visualizing Impossible 3D Worlds Using Portals

Luca Silva
*Voxar Labs, Centro de Informática*
*Universidade Federal de Pernambuco*
Recife, Brazil
lams3@cin.ufpe.br

Lucas Valença
*Voxar Labs, Centro de Informática*
*Universidade Federal de Pernambuco*
Recife, Brazil
lvrma@cin.ufpe.br

Arlindo Gomes
*Voxar Labs, Centro de Informática*
*Universidade Federal de Pernambuco*
Recife, Brazil
agsn@cin.ufpe.br

Lucas Figueiredo
*Voxar Labs, Centro de Informática*
*Universidade Federal de Pernambuco*
Recife, Brazil
lsf@cin.ufpe.br

Veronica Teichrieb
*Voxar Labs, Centro de Informática*
*Universidade Federal de Pernambuco*
Recife, Brazil
vt@cin.ufpe.br

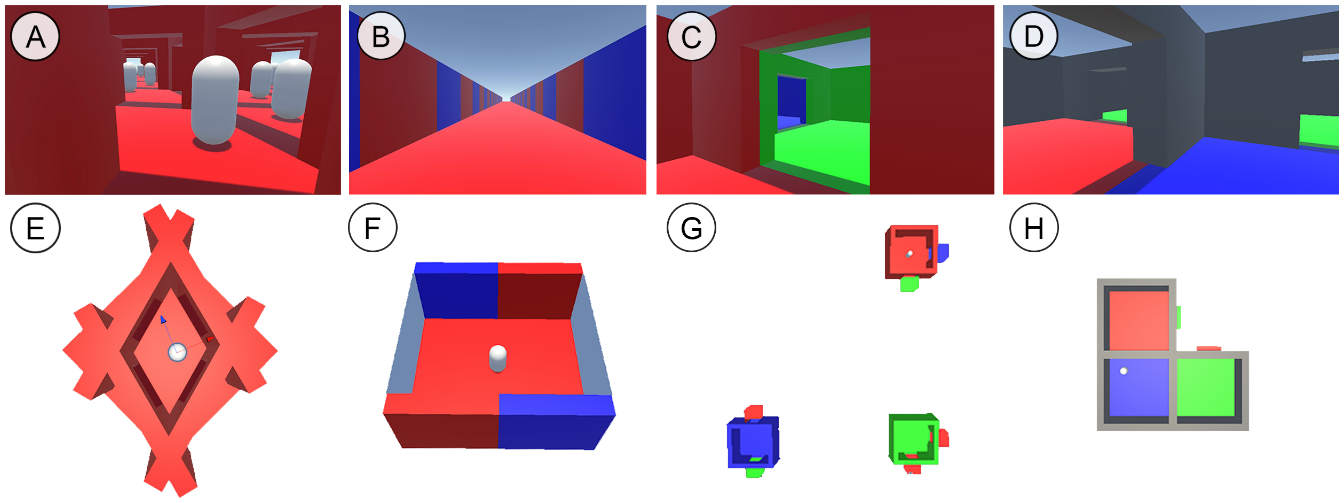

Fig. 1. Some use cases of GoThrough's portals. Top row shows the player's view, bottom row shows the third person view of the maps. A-E) An infinite mirror house with 4 portals that see each other. B-F) Infinite corridor (player from F is invisible in B's first person view in order to achieve the desired effect). C-G) Unsolvable maze with portal connections that shuffle randomly upon being crossed. D-H) A square room 3 corners, creating a sense of confusion when the player walks around.

***Abstract*—Portals are commonly used in video games (e.g., games like Portal, Antichamber, and the classic Asteroids). In this work we introduce GoThrough, a tool that enables users with little to no previous knowledge to add transformative portals to 3D scenes in the Unity game engine. We map the existing literature in portals, both in terms of academic works and web resources, as well as entertainment usages. Then, we introduce an approach for portals to work robustly both in terms of geometry and rendering, and explore common pitfalls (as well as how to handle them). The tool is shown to work in a variety of example scenarios, and has been evaluated quantitatively for performance, providing real-time performance in a variety of scenarios. User tests have also been conducted in order to analyse GoThrough qualitatively. With a SUS score of 87.5, we concluded that GoThrough is intuitive enough to be used by non-experts, making the process of creating impossible 3D worlds much less cumbersome.**

***Index Terms*—portals, scene composition, development tools, spatial distortions, Escher-like worlds**

## I. Introduction

Virtual transformative portals have been used extensively in digital applications, specially in the video game industry. A transformative portal is a portal that applies a 2D or 3D transformation to an object's rigid body, changing its location, rotation, or both. The most well known examples focus on using portals for game-play or interaction purposes. One of the earliest works, Asteroids (1979), used 2D portals to make the map continuous at the edges. A similar kind of 2D portal concept has then been used to simulate UI interactions and the exchange of documents between users [1].

Portals in 3D worlds have also been around for a while. For example, Epic's Unreal made use of impossible 3D portals to transport users all the way back in 1998. Also in that year, the game Thief utilized invisible portals to help propagate sound in 3D virtual space. This concept has been further explored in 2007 by Foale and Vamplew [2]. Even earlier than Unreal and Thief, in 1997, the game Prey had a custom-built engine that was portal-oriented and used to create scenarios that would visually confuse and challenge the player. Later, Prey's developers stated that making an engine around portals was a mistake, as there are too many caveats to watch out for. This statement is one of the reasons the proposed work, GoThrough, is implemented as a plug-in to an already consolidated engine instead of a standalone tool.

The complications that the developers of Prey were referring to might include illumination and physics issues. The problem is that as illumination and physics algorithms in games became more complicated and realistic, making portals look and feel good became more challenging. In terms of physics, transporting a player abruptly between two spaces might have many complications if considering aspects such as movement speed, infinitely-growing acceleration, gravity changes, and medium density changes, for example. In terms of illumination, ray tracing-based approaches work great for portals, as the light ray goes through the portal creating coherent illumination. Yet, for the current more common and accessible real-time illumination techniques, portals can be tricky to make look good (see Section V).

Due to the issues mentioned above, portals became less popular in the mainstream video-game industry. Yet, more recently, with more powerful hardware and engine modularity improvements, games have been adding portals back as part of puzzles or as a way to trick the user's perception. Those have been custom-made for each game, and added on top of robust 3D engines like Unreal or Source. Some games using such concepts are Glitchphobia, Half-Life: Alyx, and The Stanley Parable. In these games, portals are smoothly blended to the environment, used to create surrealistic representations of reality (e.g., infinite corridors, mirrored worlds, or impossible mazes). Other cases (e.g., the Portal franchise) use portals with visible frames to enable conscious user interaction.

Early $20^{th}$ century artist Maurits Escher, who famously portrayed impossible worlds in his art, was a precursor for many of these aforementioned ideas. In fact, the game Fragments of Euclid uses portals to attempt to recreate versions of Escher's work. Orbons and Ruttkay [3] have also proposed a way to render and interact with Escher-like 3D scenes.

Thus, portals in video-games are mostly used to distort reality. When one thinks of those distortions in 3D space, the usual idea that comes to mind is that of an obvious portal, such as a hole in the wall in the Portal games, which resembles the sci-fi idea of a *wormhole*. Yet, portals that go unnoticed and perfectly blend to the environment are usually the ones that offer more flexibility, testing both the user and the content creator's creativity. For example, a square room can be extended to have infinite corners, or even less than 4 corners (see Fig. 1, letter D and H), creating a sense of confusion on the user by breaking what is expected from reality. Some of these concepts have also been well discussed by Code Parade[1].

In terms of our work's contributions, we introduce a brief study of the history of portals and a round-up of common problems and solutions (including level design guidelines). To do that, we compile game-oriented knowledge from both academic (see Section II) and reputable online sources (e.g., Sebastian Lague's *Coding Adventures* series[2] and Ignis Incendio's *Multiple Recursive Portals and AI In Unity* tutorial series[3]). We hope that this will help close the existing literature gap on the subject. In addition, we also propose GoThrough: a drag-and-drop-based plug-in for Unity that acts as a 3D portal tool, using the aforementioned modern approaches to better handle the challenging aspects of portal implementation. To our knowledge, this is the first such tool made available to the academy for game content creation. In terms of evaluation, we examine both GoThrough's performance and its ease of use. For performance (see Section VI-A), we evaluate the impacts of having multiple visible portals and different recursion depths in terms of FPS, RAM, VRAM, and scene triangle count. In terms of ease of use (see Section VI-B), we have performed tests with 12 different users to evaluate GoThrough's usability in terms of the System Usability Scale (SUS) questionnaire [4]. We believe these evaluations provide much needed quantitative insight on the portal subject. Finally, in Section VI-C, we propose some mind-boggling use cases, which can serve as inspiration for content creators.

## II. Related Work

The investigation of 3D portals by researchers include its use and development in particular scenarios, ranging from applications on industry to games and entertainment.

As an early preliminary approach, Jones [5] proposed a way of separating a map (e.g., an apartment blueprint) into cells for each room, to facilitate 3D rendering. This concept established a baseline to others such as [6] and [7], who introduced a way to use these cell divisions to render mirror-like portals. These works have since influenced several architectural and cell-based approaches.

Moreover, in architectural cases, portals can serve as a window or texture [8] to perform culling in large parts of the scene that are not in immediate usage, breaking the 3D environment into smaller, portal-separated portions. Virtainer [9] applies this concept for industrial container field rendering.

Lowe and Datta [10] have explored the concept of how portals affect polygonal meshes, which is a very important aspect of rendering when dealing with game objects crossing portals. This has been further expanded in [11], which introduces a framework to construct scenes with more complex, transformative portals. This concept differs from traditional architecture scenarios because those cases do not apply rigid-body transforms to the user's avatar (i.e., change its location and rotation) when the avatar crosses portals between directly connected rooms of a blueprint (e.g., when traversing a digital house

[1]https://www.youtube.com/watch?v=kEB11PQ9Eo8
[2]https://www.youtube.com/watch?v=cWpFZbjtSQg
[3]https://medium.com/@limdingwen_66715

with portal-and-cell architectural rendering implemented). The work of Peterson [12] proposed alternatives which further increased performance for transformative purposes. Tilman [13], on the other hand, has improved this concept by studying how to handle more complex scenarios such as when portals with different shapes (e.g., cones, triangles, circles) interact with both 2D and 3D objects.

A scanline algorithm has also been proposed to render arbitrarily-shaped portals (e.g., cracks or holes in a wall) instead of the commonly-used geometrical formats (e.g., planes and ellipses) [14]. Finally, the cells-and-portals concept has been used to connect large maps over multiple machines [15].

Other interesting applications include offline portal rendering for movies [16] and 3D portals as a UI replacement [17].

## III. Method

In this paper we present GoThrough, a tool for easily creating portals in 3D scenes. GoThrough is a texture-based technique and its pipeline consists of a few steps performed every frame. The pipeline acts on two types of entities: *portals* and *travellers*. A pair of portals can be thought of as a wormhole through 3D space. Travellers are virtual objects that have been enabled to travel through those portals. To ensure more consistent results, the pipeline should be executed at the end of the game loop. This way, all modifications applied to portals and travellers on that frame can be correctly displayed. While describing the pipeline, some base concepts regarding portals must be considered.

Portals are considered to be planar surfaces with well defined *front* and *back* sides. A portal's normal vector always points towards its front side and portals can only be seen through their front side.

An entry portal $e$ is always connected to its destination portal $d$, which can be positioned anywhere in the virtual scene. For $e$, its front side is considered to be where the traveller is crossing its surface from. In $d$'s case, the front side is the one the traveller will appear at. Thus, a pair of portals $(e, d)$ represents a mapping from the points in $e$ to the the points in $d$. This mapping can be defined through the $4 \times 4$ 3D transformation matrix $M_d^e = M_d \cdot R_y(\pi) \cdot M_e^{-1}$, where $M_n \in \mathbb{SE}(3)$ denotes the $4 \times 4$ matrix containing an object $n$'s 3D rigid body transform composed of rotation and translation, $M^{-1}$ denotes the inverse matrix of $M$, and $R_y(\pi)$ represents a $\pi$ radians rotation around the $Y$ axis.

A portal's destination is a matter of choice. The tuple $(e, d)$ merely says $e$ leads to $d$, and the opposite ($d$ leading to $e$) is not necessarily true. $e$ and $d$ may even be the same portal if correctly configured. As some additional care is needed for those special cases (see Section V), the one-way scenario with two portals positioned at distinct locations is assumed in this explanation.

Finally, it is also important to explain that what a standard pinhole camera $c$ sees when looking through an entry portal $e$ is equivalent (though cropped, considering $e$'s dimensions) to what it would see if the camera $c$ were positioned relative to the destination portal $d$ like it currently is to $e$ (using $M_d^e$). This means the portal functions just like a window (when looking from $e$ to $d$). This concept is illustrated in Fig. 2.

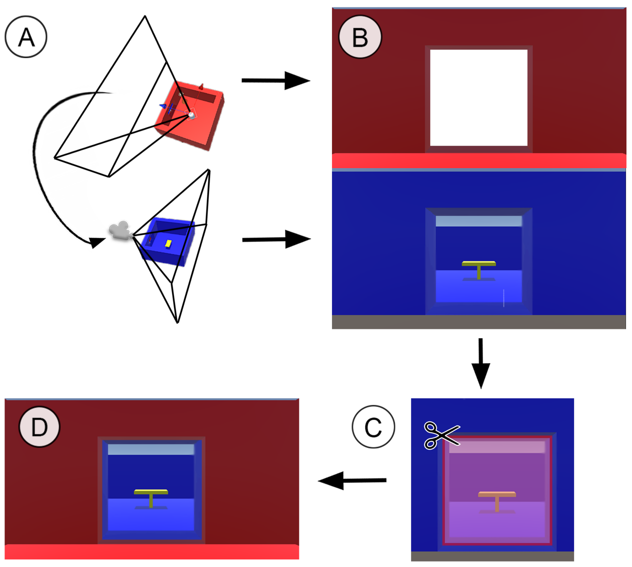

Fig. 2. Camera positioning to display portal texture. A) Scenes connected by a portal. Player Camera (looking at the entry portal in red) and GoThrough's Camera (looking at the destination portal in blue) have their camera frustrums outlined. B) First person view from both frustrums highlighted in A. C) Process of extracting the destination portal's texture as seen by the Entry Camera. D) Final result as seen by the player when looking at the entry portal.

### A. *Establishing Transitions*

In the beginning of the pipeline, it is necessary to define the set $T$ representing all *active transitions* (i.e., transitions happening at the moment). A transition is denoted as a triplet $T_\alpha = (\alpha, e, d) \in T$, composed of a traveller $\alpha$, crossing an entry portal $e$, with $d$ as it's destination.

To build $T$, our method assumes each traveller can not be part of more than one transition at a time (see Section V-A). If $\alpha$ crosses more than one entry portal at a time, only the one portal closest to $\alpha$'s present location is chosen to build $T_\alpha$. While there is a portion of $\alpha$'s 3D mesh crossing $e$'s planar surface, $e$ is considered valid for an active transition.

### B. *Teleporting Travellers*

In this step, for each active transition $T_\alpha$, a check is performed to see if $\alpha$ must be teleported. A teleport must happen when $\alpha$'s 3D pivot (mesh local origin) $\alpha_{pivot}$ crosses $e$'s surface. That is, $\alpha_{pivot}$ was on $e$'s front side at the previous frame and is now on $e$'s back side.

To effectively teleport $\alpha$, we have to place it relative to $d$ as it was to $e$. This can be achieved by applying $M_d^e$ to $\alpha$'s local-to-world matrix $M_\alpha$. By teleporting $\alpha$ we also substitute $T_\alpha = (\alpha, e, d)$ with $T_\alpha = (\alpha, d, destination(d))$, because $\alpha$'s pivot is now physically on $d$'s front side and $\alpha$'s 3D mesh is intersecting $d$'s surface.

### C. *Prepare Transitions for Rendering*

Having all travellers in position and all transitions properly tracked, it is time to ensure a traveller $\alpha$ crossing from portal $e$ to $d$ will be properly rendered on both sides. That is, if $\alpha$ is half-way through $e$, the part of its 3D mesh already across $e$ should should be rendered in front of $d$ instead.

To achieve this, $\alpha$'s mesh is rendered two times: one in its original position relative to $e$ and another relative to $d$. This step is performed through a clone $\overline{\alpha}$ positioned using $M_d^e$. A clipping operation is performed to hide the part of $\alpha$'s 3D mesh that has already crossed $e$. This hidden mesh portion is the visible part of $\overline{\alpha}$'s mesh, rendered on $d$'s side.

This operation is performed by a special shader assigned to both $\alpha$ and $\overline{\alpha}$. The shader applies an *alpha clipping* operation to discard fragments of both meshes. A clipping plane defined defined in world space is used for this. For $\alpha$, the plane is equivalent to $e$'s front side. For $\overline{\alpha}$, $d$'s front side. The result of this process is illustrated in Fig. 3. After those steps, travellers are ready to be correctly rendered by cameras.

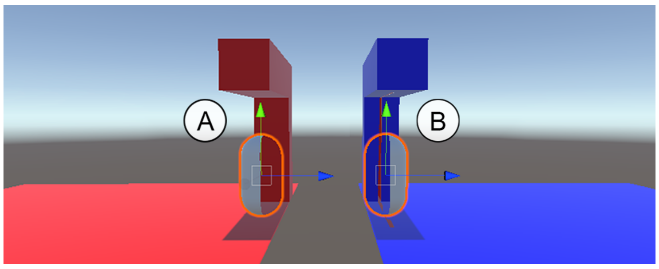

Fig. 3. Alpha clipping process. A) The traveller $\alpha$ with only it's portion in the front side of the entry portal (in red) being rendered. B) Mesh $\overline{\alpha}$ ($\alpha$'s clone) whose only portion being rendered is the one in front of the destination portal (in blue).

### D. *Rendering*

This step is responsible for the actual rendering of a scene in GoThrough. It should be executed once for every camera that is part of the pipeline.

*1) Preventing Near-Plane Clipping:* When a portal $e$ is rendered by camera $c$, and $c$ is too close to it, near-plane clipping can occur, preventing the portal to be rendered correctly. To avoid this issue, a mesh $m$ is placed behind $e$. This mesh is a 3D extension of the portal's shape (e.g., a parallelepiped for rectangular portals), and uses the same shader as $e$ (the walls of $m$ show the other side of the portal as well). All its normals are pointing inwards and its dimensions are the same as the portal's height and width, with depth $g$ equal to the length of the diagonal of $c$'s near-plane. Thus, whenever near-plane clipping occurs on $e$'s front side, $m$ is rendered instead. For an illustration, see Fig. 4.

*2) Assembling the Visibility Tree:* Before rendering what's being seen by a camera $c$, a visibility tree $\Psi$ must be assembled (as illustrated in Fig. 5). This tree accounts for portals seen inside of other portals (or inside themselves). A node $(e, d, M) \in \Psi$ represents an entry portal $e$ (with destiny $d$) seen by $c$ positioned at $M$. The root node of $\Psi$ contains the

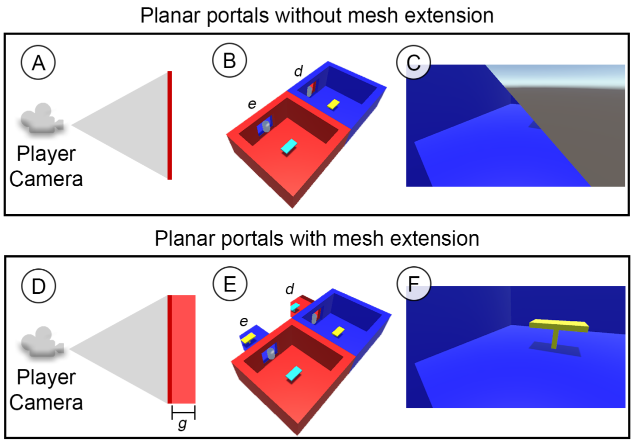
Planar portals without mesh extension
A
Player Camera
B
d
e
C
Planar portals with mesh extension
D
Player Camera
g
E
d
e
F

Fig. 4. Process to prevent near plane clipping upon crossing portals. A) Diagram of the player's camera looking at a portal with no extension mesh. B) Third person view of the scene. C) First person view of the player's camera when looking through the entry portal and suffering from clipping. Part of the background outside the scene can be seen. D) Same diagram as A, but with an extension mesh. E) Same as B, but with extension mesh rendered behind portals. F) Final result as seen by the player, with near plane clipping avoided.

camera's original rigid-body transform. The recursive step for every node is performed by creating a new child node for every entry portal $e$ visible by $c$ at the transform in the node triplet's third element. The end of the recursion occurs when there are no visible entry portals left or when a maximum recursion *depth* is reached. This process is described in Algorithm 1.

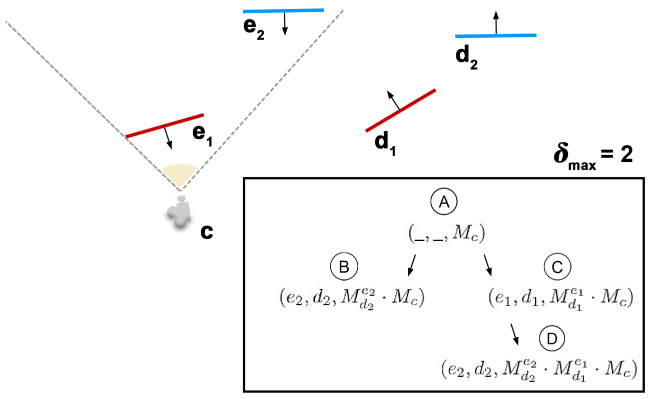
$e_2$
$d_2$
$e_1$
$d_1$
c
$\delta_{max} = 2$
A
$(\_, \_, M_c)$
B
$(e_2, d_2, M_{d_2}^{e_2} \cdot M_c)$
C
$(e_1, d_1, M_{d_1}^{e_1} \cdot M_c)$
D
$(e_2, d_2, M_{d_2}^{e_2} \cdot M_{d_1}^{e_1} \cdot M_c)$

Fig. 5. Diagram representing a sample scene with a camera $c$ and two pairs of mutually connected portals $(e_1, d_1)$ and $(e_2, d_2)$ and the corresponding visibility tree of max depth 2.

*3) Scene Rendering:* Once the visibility tree $\Psi$ is assembled, the scene can be rendered. Portals are rendered using a special shader that draws a simple texture. How those textures are rendered and how the shader samples them is what makes results look believable.

A texture for a portal $e$ seen by camera $c$ at a certain pose $M_c$ is rendered by GoThrough's camera $c_{view}$ placed using $M_d^e \cdot M_c$, where $d$ is $e$'s destination. This rendered texture represents what a camera would see through the portal, but it

**Algorithm 1:** Build Visibility Tree

```
Input:   $P$, the set of active portals
         $c$, the camera to be rendered
         $\delta_{max}$, the maximum recursion depth
Output:  $\Psi$, the visibility tree of $c$

/* creating tree nodes recursively */
Function AddChildren($\Psi$, $(e, d, M_{in})$, $\delta$):
    $node \leftarrow (e, d, M_{in})$
    /* maximum tree depth */
    if $\delta \geq \delta_{max}$ then
        return
    end
    /* iterate every active portal */
    for $p \in P$ do
        /* $e$ can't directly see $d$ */
        if $p = d$ then
            continue
        end
        /* viewpoint to render entry portal */
        $viewPose \leftarrow M_d^e \cdot M_{in}$
        /* check visibility and insert node */
        if $CanSee(c, viewPose, p)$ then
            $n \leftarrow (p, destination(p), viewPose)$
            $Insert(\Psi, node, n)$
            $AddChildren(\Psi, n, \delta + 1)$
        end
    end
    return

$\Psi(0) \leftarrow (\_, \_, M_c)$                  // root node
$AddChildren(\Psi, \Psi(0), 0)$               // begin recursion
return $\Psi$                                 // return tree
```

has to be cropped to display only what's inside the portal frame (as per Fig 2). To achieve this, portal shaders sample their textures in *screen-space* (see Fig. 6), opposed to the traditional object-space sampling (using the mesh's UVs). Portal textures should have the same aspect-ratio of $c$'s output, so that both can be perfectly aligned when displayed in screen-space.

In addition to cropping the texture to fit the portal's frame using screen space shading, we must also only render what's in front of the destination portal (not what's between the camera and the portal's back). To achieve this, the near plane of $c_{view}$ is aligned with $d$ using an *oblique view frustum* as proposed by [18]. This process is illustrated in Fig. 7.

Finally, the proper rendering of those textures is achieved by performing a depth-first traversal through $\Psi$. This way, each node will be rendered only after all of its children have been as well. Since a portal's texture depends on its observer, after a node $n$ in $\Psi$ is done with its rendering, it must restore all its children's textures to what they were before $n$'s texture was rendered. This is necessary because the children's textures would now be rendered with relation to $n$, which is

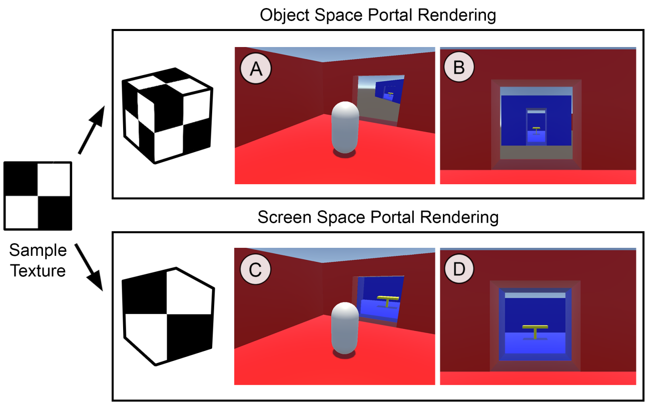

Fig. 6. Illustration of how portal textures are rendered correctly using the screen space approach. A) Third person view of a player looking at a portal rendered with the full view from GoThrough's camera. B) First person view of A. C) Same as A, but with screen space cropping of GoThrough's camera view to display just the area inside the destination portal. D) First person view of C. For details on GoThrough's camera positioning, see Fig. 2.

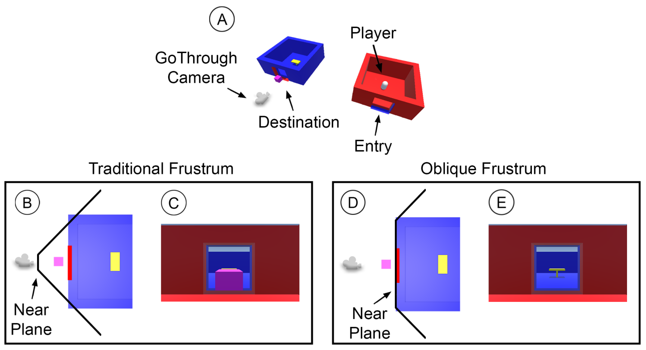

Fig. 7. Illustration highlighting the oblique view frustum's usage. A) Third person view of scene setup. B) A diagram of GoThrough's camera near plane without using the technique. C) The resulting render of B as seen by the player. Objects behind the destination portal are rendered to the entry portal. D) Same as B, but with an oblique view frustum applied. E) Final result as seen by the player. Objects behind the destination portal are correctly culled.

not necessarily coherent with $n$'s parent nodes. For instance, taking the visibility tree in Fig. 5 as an example, assuming the left-most children of a node are visited first, the visiting order would be $B \rightarrow D \rightarrow C \rightarrow A$. If $e_2$'s previous textures were not restored, since $e_2$'s texture is last modified when rendering $D$, the result produced by $B$ would be lost and the root node $A$ would be rendered with an inaccurate view of $e_2$. The entire process can be visualized in Algorithm 2.

## IV. Implementation Details

GoThrough was implemented as a plug-in for the popular 3D engine Unity. Thus, it can be easily added to video-game projects. In this section, our architecture and some Unity-specific details are described.

A core resource used in our implementation is the *Render Texture*, Unity's object oriented approach on *off-screen*

**Algorithm 2:** Render Scene

```
Input:  $c$, the camera being rendered
        $\Psi$, the visibility tree
        $\delta_{max}$, the maximum recursion depth

/* render a tree node by recursively visiting
   its child nodes */
Function RenderNode($(e, d, M)$):
    $node \leftarrow (e, d, M)$
    $\gamma \leftarrow$ new dynamic array to store old textures
    /* render child nodes (recursive step) and
       save their previous textures */
    for $n \in Children(\Psi, node)$ do
        $Insert(\gamma, RenderNode(n))$
    end
    /* update for the current input viewpoint */
    $t_{new} \leftarrow$ empty texture
    $c_{view} \leftarrow$ new camera clone of $c$ at pose $M$
    $SetObliqueViewFrustum(c_{view}, d)$
    $SetRenderTarget(c_{view}, t_{new})$
    if $Depth(\Psi, node) = \delta_{max}$ then
        /* base of recursion (e.g., a blank
           texture or a pre-determined image) */
        $RenderDefault(t_{new})$
    else
        /* render new texture coherent with
           current input viewpoint */
        $Render(c_{view})$
    end
    /* set child textures back to original, so
       they can be reused by other tree nodes */
    for $(p, t^p_{old}, t^p_{new}) \in \gamma$ do
        $SetTexture(p, t^p_{old})$
        $Release(t^p_{new})$
    end
    /* update node texture and store previous */
    $t_{old} = GetTexture(e)$
    $SetTexture(e, t_{new})$
    return $(e, t_{old}, t_{new})$

$\Gamma \leftarrow$ new dynamic array to store final textures
/* render every node but root */
for $n \in Children(\Psi(0))$ do
    $Insert(\Gamma, RenderNode(n))$
end
/* render according to root */
$Render(c)$ for $(p, t^p_{old}, t^p_{new}) \in \Gamma$ do
    $SetTexture(p, \_)$
    $Release(t^p_{new})$
end
return
```

*rendering*. Render Textures are objects that represent a GPU *framebuffer*. As such, they are costly to initialize and release, demanding careful utilization. Those objects are used to implement the portal textures described in Section III-D3. To reduce the portals' computational cost and better manage system resources, a Render Texture pool was implemented to re-utilize objects, avoiding unnecessary memory reallocation.

The second important Unity resource used is the *Prefab* system. Prefabs are serialized, reusable game objects. Those objects can be promptly referenced in Unity scenes through drag-and-drop. Our implementation uses Prefabs to provide ready-to-use portals. All one has to do to create a pair of connected portals in a scene is to drag and drop two portal Prefabs and link them as each other's entry and destination through the Unity Editor. Further modifications to these portals (e.g., scaling and mesh modifications) can also be performed by the user.

Our implementation performs additional culling during the visibility tree assembly step (see Section III-D2). This culling is performed when checking a portal's visibility from the viewpoint of another portal. When rendering an entry portal $e$, we only need to render what can be seen through the frame of $e$'s destination portal $d$. As described in Section III-D3, what is seen by $c_{view}$ (GoThrough's virtual camera at $d$'s side) is cropped in screen-space to be displayed on $e$'s surface. Yet, that cropping happens after the entire $c_{view}$ camera frame is already rendered. Thus, to avoid the costly, recursive rendering of portals that would be entirely cropped by the screen-space rendering, an intersection test is run on the 2D bounding boxes of $d$ and any portal $p$ visible by $c_{view}$. If there is no intersection, $p$ is not rendered.

Our architecture is based on Unity *Components*, adding behaviors to existing game objects. Our implementation is written in C# and consists of the following classes:

- **Portal**: A Component representing a portal;
- **Traveller**: A Component representing a traveller;
- **PortalRenderer**: A Component representing a camera object rendered through our pipeline. Objects of this class are responsible for the actual execution of the Render step of the method described in Section III-D;
- **VisibilityTree**: A class representing the visibility tree described in Section III-D2;
- **RenderTexturePool**: A class representing a pool of Render Textures. Objects of this class are responsible for managing textures used by PortalRenderers. As previously explained, by using pools, Render Textures are created and released less frequently, allowing for better performance.

Our Unity plugin implementation is publicly available and free for academic usage[4].

## V. Portal Pitfalls & Guidelines

In this section, we provide guidelines on how to handle limitations derived from the use of portals in virtual environments

[4]https://github.com/lams3/GoThrough

and, in particular, from the texture-based approach.

### *A. Multi-Portal Travelling*

As described in Section III-A, our method assumes a traveller $\alpha$ to be intersecting at most one portal at a given frame. This occurs due to the necessity of keeping a clone $\overline{\alpha}$ to represent it on the other side of the portal (see Section III-C). It is possible to define multiple clipping planes for a traveller, as well as position multiple copies of it (which should also have more clipping planes) as more transitions occur. Those copies could also recursively account for intersections with them. This issue can quickly scale and become costly, therefore GoThrough opts not to consider transitions other than the closest one. The resulting behavior can be seen in Fig. 8 letter D.

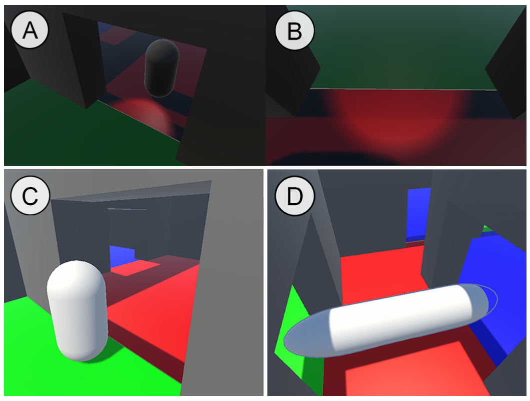

Fig. 8. Failure cases. A) Third person view of a player in a dimly-lit scene with a flashlight pointed at a portal. The flashlight beam is not correctly rendered across the portal. B) First person view of A. C) Third person view of a player in a bright scene with point lights. The player's shadow is not properly rendered across the portal. D) A large object outlined while it crosses multiple portals at once. The back part of the object is not correctly rendered.

While considering this issue, the guideline for designing virtual environments with GoThrough is to place portals with enough space between them so that a traveller is never crossing more than one portal at a time.

### *B. Lighting*

While aiming for illusory effects, the design of the scene must be done carefully regarding the positioning of portals and lights. Given our technique does not account for light transfer between portals, there can be discrepancy (as per Fig. 8 letters A, B and C) in the shading near portals. Strategies to avoid such discrepancies are:

- **Remove Lighting**: The game Antichamber makes heavy use of this approach. No lighting calculations are performed on the objects of the game, avoiding discrepancy.
- **Perpendicular Directional Lights**: If a directional lights is perpendicular to portals, there is no direct light transport occurring through the portal. This approach can be used to avoid discrepancy while keeping some liberty in the use of lighting.
- **Point/Spot Light Control**: If any point or spotlight is used, their range should not reach to a portal. If such lights don't have enough range to reach a portal, they will still be correctly rendered.

### *C. One-way & Mirror-like Portals*

As mentioned in Sec. III, each portal $e$ is connected to it's destination $d$. Some care is needed in special cases though. Those cases are:

*1) One-way Portals:* This occurs when $destiny(d) \neq e$, that is, $e$ leads to $d$, but $d$ leads elsewhere. This is supported by our tool, but can lead to strange behaviour due to it's definition.

If a camera $c$ passes through $e$ while looking at it, nothing is noticed, a smooth transition occurs as expected. But if $c$ passes through $e$ looking back, it will instantly see $d$'s destination rendered in $d$ when teleported, ruining the illusion.

Also, for a spectator looking at $e$ when a traveller $\alpha$ is crossing it, there is nothing unusual happening, since $\alpha$ will be properly rendered on both sides. But if a spectator looks at $d$, it will see only the portion of $\alpha$ that has already crossed the portal, and since $d$ leads elsewhere but $e$, $\overline{\alpha}$ will not be seen by the virtual camera rendering $d$.

Therefore, although this type of portal is supported by our tool, they should be used with care, given that visual artifacts may occur from it's usage.

*2) Mirror-like Portals:* A portal $e$ whose destination $d$ is itself, that is $e = d$, can be easily defined in GoThrough, but due to the $R_y(\pi)$ used to define $M_d^e = M_e^e$, it will behave as a *non-reversing mirror*. To achieve the traditional mirror effect, an flip on the z axis $S_z(-1)$ should be used instead. However, GoThrough does not implement this feature.

## VI. Experiments and Results

### *A. Performance*

In order to evaluate GoThrough's performance, we have conducted tests using a laptop running Windows 10 with a quad-core CPU @ 2.5 GHz, 8 GB of RAM, and an NVIDIA GTX 950M GPU with 2 GB of VRAM. All tests were conducted in the 3D scene from Fig. 1 (letter F), and output frames were rendered at resolution $1920 \times 1080$. The scene overall had 1052 triangles, most of these being from the player's capsule, which for this experiment was not invisible as shown in Fig. 1 (letter B).

The runtime per frame of GoThrough is mostly taken up by rendering, which occurs by traversing recursively the visibility tree structure described in Section III-D2. The size of the tree is determined by two main factors: how many portals are visible in each recursion(tree width), and what is the maximum recursion depth allowed (tree height).

We conducted two separate experiments on the scene to assess the performance impact of each stated factor. First, we fixed the amount of visible portals as 1 (with the destination portal behind the player, not directly visible), and increased the maximum allowed recursion depth from 1 to 64, in powers of 2. This caused the tree to increase only in height. This will be further refereed to as the *depth experiment*.

For the second experiment, we kept the maximum recursion depth at 1 (so that portals are only rendered if they are directly visible by the player's camera) and gradually added portals on top of each other (overlapping), 2 portals at a time, one in front and one behind the player. This way, we had 1 new visible portal per pair. The amount of visible portals was increased from 1 to 64 as well, in powers of 2. The result was a tree with as many nodes as in the first experiment, but with the same width as the depth experiment's tree's height. This will be further refereed to as the *portals experiment*.

The amount of tree nodes and rendered triangles per scene can be seen in Table I. Note that for each node in the tree, the triangles in the scene have to be rendered again. This fact must be considered when developing high-poly environments with portals.

TABLE I. Details on the amount of nodes in the visibility tree and the amount of triangles rendered for the performance experiments.

| | Amount of Visible Portals or Maximum Recursion Depth | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| # Nodes | 1 | 2 | 3 | 5 | 9 | 17 | 33 | 65 |
| # Triangles (in thousands) | 1.0 | 3.1 | 4.1 | 8.2 | 14.3 | 27.6 | 55.3 | 108.5 |

The results of both experiments in terms of RAM, VRAM, and processing time per frame (in ms) can be seen in the plot from Fig. 9. The execution time in the portals experiment showed to be similar to the ones in the depth experiment for lower numbers of portals. However, the impacts from 16 portals onward showed a significant loss of performance if compared to the same number of recursion depth (and therefore, the same size of the visibility tree). This performance gap is likely related to the fact that the graphics card had 2 gigabytes of VRAM available, which was quickly taken up by the multiple portal textures in the portals experiment. As the plots show, execution time increases simultaneously with RAM utilization, which the system uses as a replacement when VRAM runs out and is much slower. Modern graphics cards with higher memory may not suffer similar issues, being able to display more portals simultaneously.

The VRAM and RAM of the recursion depth experiment remained constant (as there are no new portal textures being instantiated due to our Render Texture pool, see Sec. IV). Thus, the depth experiment shows linear scalability in execution time, as expected (note that the chart is log-log, which is why the curve looks exponential).

Additionally, we measured how many Render Texture objects were allocated, to confirm the usage of VRAM by portal textures. Results can be seen in Fig. 10. Here we show that the amount of textures allocated in the portals experiment scales linearly with the amount of portals (while it remains constant in the depth experiment). With the increased number of textures, the number of swaps between them heavily increases. Once all VRAM was taken up by those textures and RAM was allocated instead, those swaps became a performance bottleneck due to the use of slower memory.

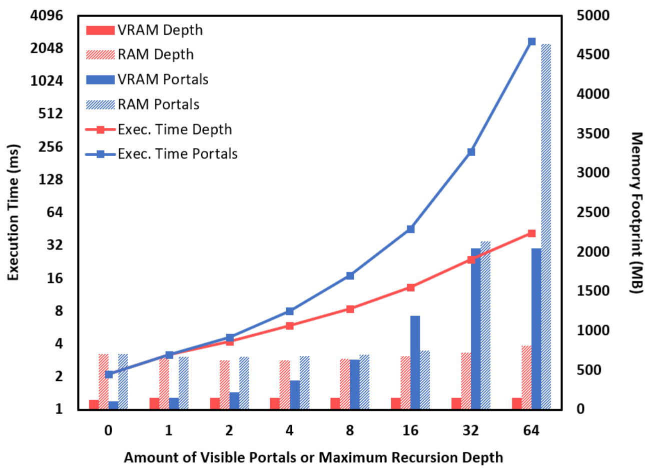

Fig. 9. Plot displaying the RAM/VRAM footprint (log scale, right y-axis) and ms (left y-axis) taken to render a frame as the amount of portals or recursion depth increase exponentially.

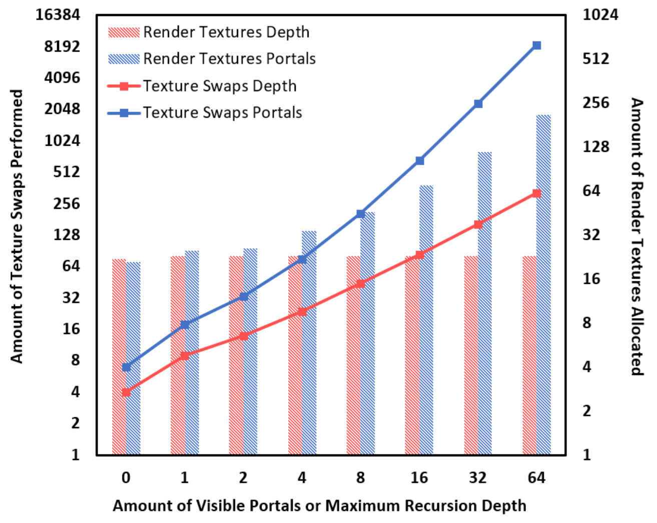

Fig. 10. Log-log plot showing how many Render Textures are allocated (right y-axis) and how many texture swaps are performed (left y-axis) as the amount of portals or recursion depth increase exponentially.

## B. User Experiments

To understand how users with different experiences handle our tool we conducted a user experiment. Users were interviewed individually and later asked to respond to a SUS evaluation [4] questionnaire. Each interview consisted of three steps:

- **Portals and Tool Presentation**: First, to provide basic knowledge about portals, the concept was briefly introduced and some common use-cases were presented. GoThrough was also presented as a tool proposing to facilitate the process of creating portals in a 3D scene.
- **Guided Experiment**: In this step, the user was asked to perform a series of tasks adding portals in a specific

manner to a functional game scene.

- **Free Use Experimentation**: The user freely explored our tool, to reproduce any desired behavior.

On the guided experiment the user reproduced the behaviors illustrated in Fig. 11. The specific tasks performed during this step of the experiment were:

1) Creating an Unity project;
2) Importing GoThrough's package;
3) Opening the test scene where modifications would be done;
4) Setting the player game object as a traveller;
5) Setting the player's camera to properly render GoThrough Portals;
6) Positioning a pair of portals on the scene;
7) Test the resulting scene.

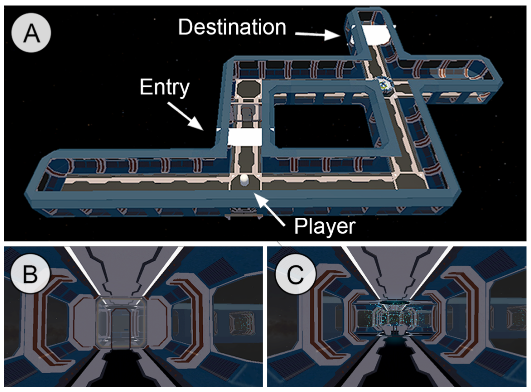

Fig. 11. Test scene used during user tests. A) Third person view of the map (with ceiling disabled for visibility purposes). B) Player's first person view with portals disabled. C) Same as B but with portals enabled.

The experiment was applied upon a group of 12 users with ages from 22 to 38 years old. All participants were students from Computer Science and Computer Engineering in BSc, MSc and PhD levels. They informed their experience level with Unity as one of the following groups: First-Timers (2 users), New Users (4 users), Moderate (4 users) and Experienced (2 users).

Performance errors related to the portals occurred with four users that tried to put more than three portals in front of each other using five levels of recursion, causing a massive performance demand for their computers.

The overall score on SUS was 87.5 points (see Fig. 12), which classifies our pipeline's usability as A+. In particular, experienced users of Unity gave lower scores on questions regarding "*confidence on use*" and "*needed help to use*" (see Fig. 13). We attribute such scores to the fact that GoThrough a the time of testing did not contain yet a detailed documentation and README file, which is the main source of information these users are used to request when exploring new tools and features on Unity.

During their free time, five users presented very innovative uses of our portals. One positioned portals in a way that

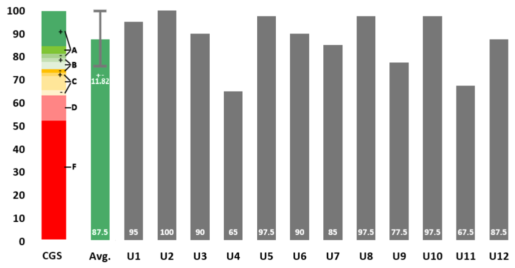

Fig. 12. System Usability Scale (SUS) average score for each of the 12 interviewed users. Last column (from left to right) shows the average of all responses, with standard deviation. The first column shows the Curved Grading Scale (CGS) [19], which can help interpreting SUS scores.

the player only had a squared cyclical path, which visually appeared to always be pursuing himself. The other one created a "*house of mirrors*" effect, where he could see many copies of the characters but struggles to find the way out. Two users created "*infinite corridors*" using two portals, one in front of each other. Another experimented with gravity, placing a portal $e$ horizontally and a portal $d$ vertically, but this mechanic needed to be further implemented.

## C. *Additional Use Cases*

Finally, to demonstrate GoThrough's capabilities, four additional use cases were developed (as per Fig. 1). Those scenarios aim to provide insights mechanics achievable through the use of our tool.

*1) Three-Corner Room:* This case shows how basic portal placement can achieve impossible geometries on the player's perception with little effort. As per Fig. 1, the scene consists of an L-shaped room with a pair of mutually connected portals linking the red and green sections of the room. This creates the illusion that the room is shaped like a (impossible) regular polygon with three $90\,\text{deg}$ angles. The effect can be further extended to contain five or more $90\,\text{deg}$ angles, all one has to do is create more rooms and connect them through portals.

*2) House of Mirrors:* In this case an interesting effect is achieved through portal recursion. Fig. 1, shows the scene layout, consisting of four portals positioned as the sides of a rhombus. Each portal is connected to itself. Thanks to recursion, the player is rendered many times inside those portals, provoking a sensation similar to the one of the "*house of mirrors*" common in amusement parks.

*3) Infinite Corridor:* This case also makes use of recursion. This time, a pair of mutually connected portals is placed in opposing sides of a corridor (see Fig. 1). To achieve the desired effect, an extra modification is done, in which GoThrough's camera is configured to don't render the player's mesh. Also, by using a higher recursion depth (this can be easily done since only one portal is directly visible at a time, see Section VI-A for more), the scene is rendered many times, creating the illusion of an infinite space.

*4) Unsolvable Maze:* With this case, we explore a possible mechanic where the player is put on a infinite, unsolvable

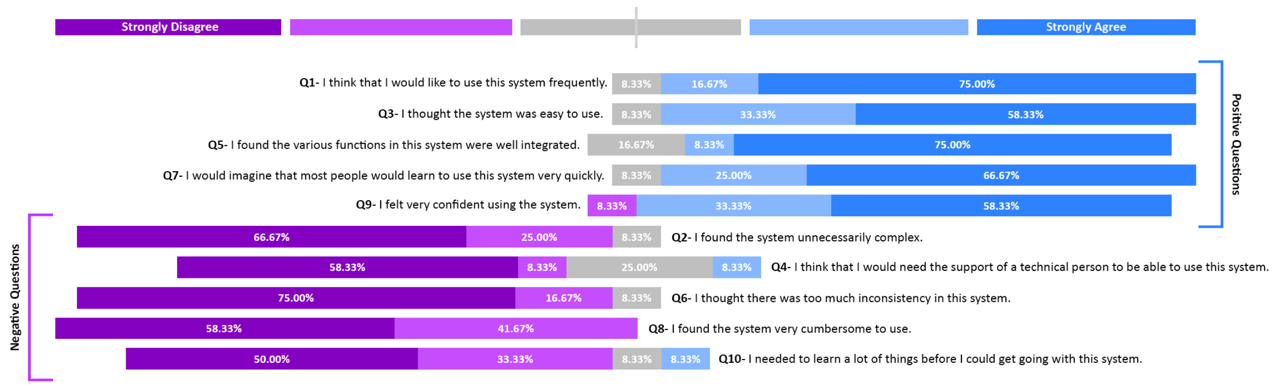

Fig. 13. System Usability Scale (SUS) answer percentage per question from the 12 users interviewed after using GoThrough.

maze. To achieve this the level is subdivided into cells (see Fig. 1), with each cell containing a few portals. Also, a script was created to modify the destinations of the cells' portals once the player is teleported. In this script, the cell the player was teleported from and the cell the player was teleported into are not modified, but all other cells are. This creates the feeling of an ever-changing maze, in which the path a player made from cell $A$ to cell $B$ may not exist anymore once it's completed.

## VII. Conclusion

In this work we have introduced GoThrough, a plug-in that enables the creation of impossible 3D worlds in the popular game engine Unity. We have detailed intrinsic aspects of portal concepts, related pitfalls and implementation details. Additionally, we quantitatively evaluated how system performance is affected by portal aspects, as a way to provide guidelines for content creators that intend to use such concepts in their work

Finally, we have also addressed the issue of portals being cumbersome to implement related to the numerous caveats to watch out for when attempting to create a pleasant user experience. GoThrough was also evaluated by users, which rated it with 87.5 points out of 100 on the SUS Score. Besides accomplishing a designed task, users showed to be capable of making creative uses of GoThrough, conceiving diverse scenarios with different mechanics (see Fig. 1).

As future work, we intend to perform more usability tests on a wider, more diverse user base. We would also like to test GoThrough for VR, to expand virtual environments while keeping the player in a small physical area. Finally, performance can be improved by using some existing culling techniques or even using stencil buffers to render portals directly to the desired target, as this can greatly reduce memory footprint compared to our current texture-based approach.

## References

[1] S. Voelker, M. Weiss, C. Wacharamanotham, and J. Borchers, "Dynamic portals: a lightweight metaphor for fast object transfer on interactive surfaces," in *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, 2011, pp. 158–161.
[2] C. Foale and P. Vamplew, "Portal-based sound propagation for first-person computer games," in *Proceedings of the 4th Australasian conference on Interactive entertainment*, 2007, pp. 1–8.
[3] E. Orbons, Z. Ruttkay *et al.*, "Interactive 3d simulation of escher-like impossible worlds," *Journal of Vacuum Science and Technology*, 2008.
[4] J. Brooke, "Sus: a "quick and dirty'usability," *Usability evaluation in industry*, p. 189, 1996.
[5] C. B. Jones, "A new approach to the 'hidden line'problem," *The Computer Journal*, vol. 14, no. 3, pp. 232–237, 1971.
[6] J. M. Airey, J. H. Rohlf, and F. P. Brooks Jr, "Towards image realism with interactive update rates in complex virtual building environments," *ACM SIGGRAPH computer graphics*, vol. 24, no. 2, pp. 41–50, 1990.
[7] D. Luebke and C. Georges, "Portals and mirrors: Simple, fast evaluation of potentially visible sets," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, 1995, pp. 105–ff.
[8] D. G. Aliaga and A. A. Lastra, "Architectural walkthroughs using portal textures," in *Proceedings. Visualization'97 (Cat. No. 97CB36155)*. IEEE, 1997, pp. 355–362.
[9] M. Escrivá, M. Martí, J. M. Sánchez, E. Camahort, J. Lluch, and R. Vivó, "Virtainer: graphical simulation of a container storage yard with dynamic portal rendering," in *Ninth International Conference on Information Visualisation (IV'05)*. IEEE, 2005, pp. 773–778.
[10] N. Lowe and A. Datta, "A fragment culling technique for rendering arbitrary portals," in *International Conference on Computational Science*. Springer, 2003, pp. 915–924.
[11] ——, "A technique for rendering complex portals," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 1, pp. 81–90, 2005.
[12] A. Petersson, "Fast complex transformative portals," 2013.
[13] M. Tillman, "Complex transformative portal interaction," 2015.
[14] Y. Yang and H. Kang, "An improved scan-line algorithm for rendering arbitrary portals," in *Recent Developments in Intelligent Computing, Communication and Devices*. Springer, 2019, pp. 1073–1080.
[15] I. Kotziampasis, N. Sidwell, and A. Chalmers, "Portals: Aiding navigation in virtual museums." in *VAST*, 2003, pp. 149–154.
[16] P. Coleman, D. Peachey, T. Nettleship, R. Villemin, and T. Jones, "Into the voyd: teleportation of light transport in incredibles 2," in *Proceedings of the 8th Annual Digital Production Symposium*, 2018, pp. 1–4.
[17] S. Hickey, L. Arhippainen, J. H. Vatjus-Anttila, and M. Pakanen, "User experience study of concurrent virtual environments with 2d tab and 3d portal uis," in *2013 International Conference on Engineering, Technology and Innovation (ICE) & IEEE International Technology Management Conference*. IEEE, 2013, pp. 1–12.
[18] E. Lengyel, "Oblique view frustum depth projection and clipping." *J. Game Dev.*, vol. 1, no. 2, pp. 1–16, 2005.
[19] J. Sauro and J. R. Lewis, *Quantifying the user experience: Practical statistics for user research*. Morgan Kaufmann, 2016.