Overlapping Architecture: Implementation of Impossible Spaces in Virtual Reality Games

Rafael Epplée[1] and Eike Langbehn[3,2]

[1] Universität Hamburg, Germany
[2] Curvature Games, Germany
https://curvaturegames.com
[3] Hamburg University of Applied Sciences, Germany
eike.langbehn@haw-hamburg.de

Abstract. Natural walking in virtual reality games is constrained by the physical boundaries defined by the size of the player's tracking space. Impossible spaces, a redirected walking technique, enlarge the virtual environment by creating overlapping architecture and letting multiple locations occupy the same physical space. Within certain thresholds, this is subtle to the player. In this paper, we present our approach to implement such impossible spaces and describe how we handled challenges like objects with simulated physics or precomputed global illumination.

Keywords: Virtual Reality · Games · Locomotion.

1 Introduction

Virtual Reality (VR) enables immersive gaming experiences which provide more natural spatial cues than games on a 2D screen. Natural locomotion further increases the sense of presence when exploring these virtual environments (VEs) [8], but is limited by the size of the physical tracking space of the player.

Since VEs are often larger than the available tracking space, different locomotion techniques have been employed to help the user navigate these VEs. Among these are joystick-based continuous motion [12], teleportation [1], and redirected walking (RDW) [9].

Impossible spaces are another technique that leverages self-overlapping architecture to build layouts that would be impossible in the real world. By making virtual rooms partially overlap with each other, the available virtual space can be enlarged without users noticing. However, if the overlap is too large, some users will start to detect it [11]. Figure 1 shows an example room layout using impossible spaces.

By using impossible spaces, players can explore larger virtual worlds in confined real-world play spaces by natural walking which is known to be more presence-enhancing and causes less disorientation [12]. Hence, the whole player experience can be improved with impossible spaces. However, there are some technical obstacles in the implementation of this technique.

In this paper, we present our approach of implementing impossible spaces in the Unity engine. There are a couple of studies that evaluate the perceptibility of impossible spaces, but to our knowledge there is no publication that discusses benefits and drawbacks of different implementations. Section 2 describes related work in the field. Section 3 outlines our actual implementation of impossible spaces, detailing interesting edge cases and how we handled them. Section 4 concludes the paper.

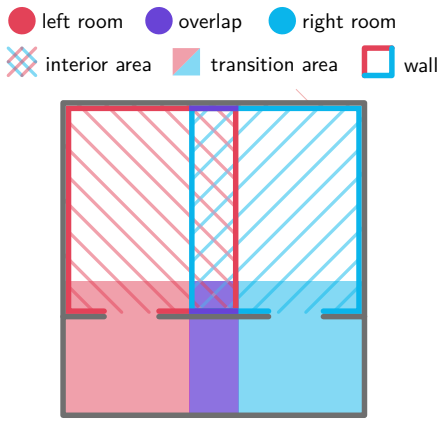

Fig. 1. An illustration of two overlapping rooms with their respective interior and transition areas.

2 Related Work

Impossible spaces were introduced by Suma et al. [11,10], who demonstrated their effectiveness in two experiments that investigated the possible amount of overlap.

In their experiments, Suma et al. reported that participants without prior knowledge of impossible spaces noticed them hardly ever, even with overlaps of 75% (which exceeds the absolute detection threshold mentioned in the experiment by far) [11]. Even when participants noticed the overlap, they only referred to them as "weird" or "strange", and none reported a negative effect on their experience.

In terms of manipulation detection, a corridor with additional turns is even more effective than a longer corridor [13], and curved corridors between the overlapping rooms are more beneficial than right-angled corridors [14].

In addition, there were also approaches to generate overlapping rooms automatically at runtime, called flexible spaces [15], and experiments that combined impossible spaces and redirected walking techniques [2].

Impossible spaces have already been used in games as well, for example by the VR game Unseen Diplomacy [6]. It arranges the (self-overlapping) rooms in such a way that the player does not recognize that she is only walking back and forth in a 3m x 4m large area. The game Tea for God even creates their impossible spaces procedurally for each run and can handle flexible tracking space sizes [7]. There is a toolkit for the Unity engine that enables procedural environment generation and manual world-building for impossible spaces[4].

2.1 Implementation with Portals

A popular technique for implementing impossible spaces are portals, virtual "windows" that teleport users to a different location when stepped through. This technique was used to great effect in the 2007 video game "Portal"[5], which made portals an explicit gameplay element used for solving puzzles. It is also the technique underlying the TraVRsal toolkit for building worlds with impossible spaces [16] which uses portals imperceptibly to maintain the illusion of a normal, non-overlapping world. Previously, non-VR games such as the 2013 video game "Antichamber" used portals like this as well to create unusual, challenging puzzles [4].

While portals promise a conceptually simple, elegant implementation of impossible spaces, they are technically complex and hard to implement. The game "Portal" and the TraVRsal toolkit implemented them using the stencil buffer [3], essentially having a fragment shader render pixels depicting portals' surfaces using a camera with a transformed location. This needs additional work to correctly simulate light and objects traveling through portals. Tricks such as duplicating light sources on both sides of the portal help, but often still leave deficiencies, such as the lack of support for light transfer when precomputing global illumination [17]. Stereographic rendering can lead to rendering artifacts that require further workarounds ([17]). Portals visible through other portals require special consideration and are usually limited in their depth. For this reason, the TraVRsal toolkit places tight constraints on the locations and number of portals, and does not support portals visible through other portals at all ([18]).

In summary, it is difficult to implement portals in a truly imperceptible way. There are many situations and interactions with other parts of the application that need to be considered, e.g. other shaders, objects controlled by players, lighting, rendering of portals visible through other portals, and more. Stylized graphics as found in the earlier mentioned game Antichamber might make it easier, making it favorable for certain scenarios. But even Antichamber contained some situations in which the limitations of the implementation became apparent, breaking the illusion of a continuous non-euclidean space ([4]). Removing other features like physics simulation from an application might make implementing portals easier as well, but this limits the technique to certain environments.

[4] https://blog.wetzold.com/2020/07/02/an-editor-for-impossible-spaces-in-virtual-reality/

[5] `https://en.wikipedia.org/wiki/Portal_(video_game)#Gameplay`

Implementations usually work by repeating the render process to render the view into a portal, which incurs a manageable, but significant performance overhead. Using portals requires overlapping rooms to be placed at a different location than the rest of the level, causing inconvenience for level designers.

3 Implementation

Besides portals, which work by imperceptibly teleporting users, there are also other methods of implementing impossible spaces. The following section outlines our approach to implement impossible spaces along with its advantages and disadvantages: manipulating room visibility, which works by dynamically hiding geometry that would reveal an overlap to users. We provide the rationale for choosing this approach for our implementation and describe the way it works, including special cases it handles and limitations of its capabilities. We explain the mechanics behind additional features, namely the handling of objects with simulated physics, precomputed global illumination (GI), and room doors. The source code is permissively licensed and available for download.[6]

3.1 Overview

Our implementation method of impossible spaces is to remove overlapping rooms from the world when the player is not near them. By keeping track of the player's current location in the world, which overlapping rooms are visible from that location, and which overlapping rooms that location is inside of, the implementation can reveal and hide the appropriate rooms at the right time.

To achieve this, the space from which an overlapping room is visible, and the space encompassing its interior, are marked by the level designer. Detecting the player entering and leaving these spaces and hiding rooms from view have a negligible performance impact, making this technique feasible in many scenarios, even with lots of overlapping rooms. Implementing this technique is straightforward compared to the portal technique. A mechanism for designating room boundaries makes no assumptions of a project's rendering pipeline and level layout, making it more flexible and easier to integrate than fragment shaders using stencil buffers.

The show-and-hide approach has some drawbacks as well. Similar to the portal technique, hiding rooms requires some additional work to correctly handle objects with simulated physics, and when precalculating global illumination using path tracing, overlapping rooms have to be considered in isolation from the rest of the level to prevent artifacts in areas where they overlap. This imposes some restrictions on developers' workflows. Additionally, level designers have to be careful to prevent situations in which users witness rooms suddenly appearing or disappearing.

While portals might be a favorable approach for some specific virtual experiences, we wanted a generic solution usable for a wide range of applications.

[6] `https://gitlab.com/raffomania/impossible-spaces`

The use of portals imposes restrictions performance budgets, render pipelines and development workflows, and easily leads to edge cases where the technique breaks down. Because the reveal-and-hide approach imposes fewer restrictions and is more robust, we chose it for our implementation.

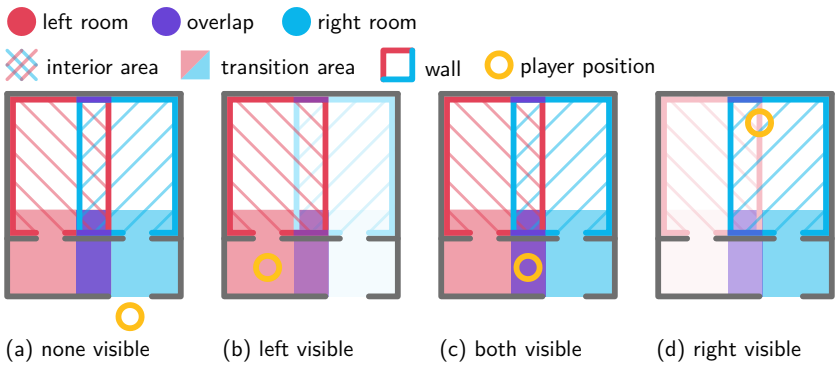

Fig. 2. Examples of different player positions relative to overlapping rooms and their respective visibility states.

3.2 Transition between rooms

In our implementation, overlapping rooms are represented by three components.

A transition area A trigger collider covering the area outside the room. This is the area in which the room should still be visible to the player. When the player leaves this area, it is considered safe to hide the associated room.

An interior area A trigger collider covering the interior of the room. When the player enters this collider, they are considered to be inside the room.

A room container An object containing everything in the room that should be hidden.

See Figure 1 for an example of two overlapping rooms with their interior and transition areas. Figure 2 shows the different visibility states of the same rooms for some player positions. To detect which area the player is currently in, an arbitrary object associated with the player is assigned a collider that triggers collision events when entering and leaving overlapping room areas. An accompanying script processes these events to keep track of the player's location, which room they are currently in and which rooms they can see, activating and deactivating them appropriately. To function correctly, it is important to distinguish when a player is inside a room and when they are outside. Simply looking at one area collision event at a time results in ambiguous situations: for example, figure 2 (d), the player is in the interior area of both rooms. To know

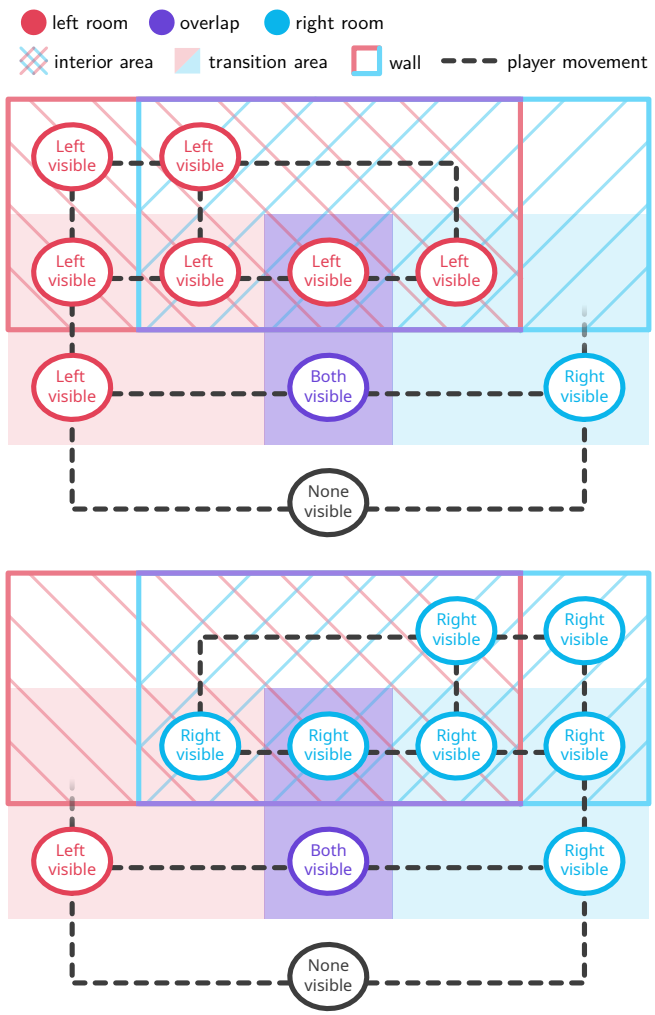

Fig. 3. The behavior of our implementation for a simple case. Execution starts at the node labeled "none visible." Depending on which room's interior players enter first, the top or the bottom graph is traversed, with each node indicating the visibility for both rooms.

which one to show and which one to hide, the implementation has to keep track of the entrance through which a player entered a room. This is why both figure 1 and 2 show the transition area overlapping with part of the interior area of the rooms. This way, the overlap between transition and interior area is treated as the entrance to the room. The implementation knows that a player has entered when they are in the interior area of a room while they are simultaneously in its transition area.

Figure 3 details the implementation's behavior in the context of a simple example. Depending on which room's interior area players enter first, one of the two graphs depicted becomes "active," determining the visibility of both rooms. As the graphs show, once players enter an interior area of a room, that room stays visible until players exit via the corresponding transition area. This behavior is a design decision coming with advantages and disadvantages which we discuss later in this section.

Note that nodes in the overlapping transition areas have less outgoing edges than most other nodes. The player position is determined by discrete events sent to the script by the Unity engine, signaling whether the player has entered or left a specific area. Since only one of these events can arrive at a time, it is not possible to enter or exit two areas at once, e.g. going directly from the "none visible" node to the node labeled "both visible." Specifically, if players moved straight between those two locations, it would result in two successive events, and for a short time only one of the two rooms would be visible.

So far, we have only covered "legal" transitions, that is, transitions where players obey the rules of the physical world when inside the virtual environment. However, in VR, players can move through walls. This means that figure 3 doesn't show every possible transition our implementation had to handle. Figure 4 adds highlighted transitions and their resulting nodes that occur when players move their head across the defined areas in ways that are considered exceptions to the normal behavior. This doesn't necessarily have to be malicious action. Since VR can be disorienting, in some moments players might fail to recognize walls and move through them. At other times, they might be unaware of the volume of their heads, not noticing it intersecting with a wall. In these situations, hiding a room from view the instant a player moves their head outside its boundaries would only add to their confusion. Hence, we decided to keep a room always visible as long as players are inside, meaning as long as they didn't exit via the corresponding transition area. As figure 4 shows, this can have some unfavorable consequences. Once players are outside a room's interior area, if they move towards a different part of the world, e.g. a different room, the implementation is now "stuck," not showing any other room than the one players originally entered. However, we consider this behavior highly unlikely in real-world situations.

A solution to this problem might be to "reset" the state and show a different room once players enter its transition area. However, since transition areas might overlap with interiors of rooms they don't belong to (see figure 4), if a player is inside one room and walks into the transition area of another, this would lead to undesired behavior. As mentioned above, simply hiding rooms once players exit their interior areas would cause undesired states as well. Considering the alternative choices, we judge the current behavior to be an acceptable tradeoff.

3.3 Handling Objects With Simulated Physics

When a room gets hidden, it is temporarily removed from the game. This means that any behavioral components such as bounding boxes for physics simulation

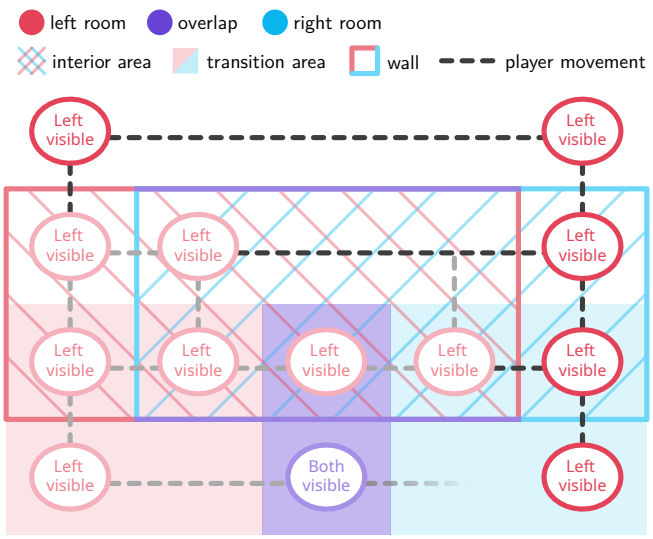

Fig. 4. The behavior of our implementation in the case that players walk through walls. Transitions that lead through walls are highlighted in deep black. States that result from these transitions are highlighted in deep red.

get disabled as well, which poses a problem for dynamic physics objects players can carry around with them. As these are not children of the overlapping room in the scene tree, they won't get hidden along with the room, and might react to the removal of the room. For example, a flashlight left on a table by a player might drop onto the floor. Additionally, objects at the overlapping space between two rooms might react to another room returning to existence.

To prevent this, each overlapping room tracks objects with simulated physics entering and exiting the room, hiding them when another room becomes visible or when the room itself gets hidden.

3.4 Precomputed Global Illumination

A popular technique for modern games is precomputed global illumination (GI). Performance-intense ray tracing computations are executed during development, before running the application. They are encoded in "lightmaps," textures that allow performance-efficient rendering of indirect lighting at runtime. This process is also known as "baking" lightmaps. Baking lightmaps for overlapping rooms can introduce artifacts in the areas they overlap. Since lightmaps do not get updated at runtime, when one room gets hidden from view, the indirect lighting still looks as if both rooms were visible at the same time.

To prevent this, overlapping rooms have to be baked separately. In the Unity engine, this means putting them into different scenes, since that is the only way to restrict the baking process to the subset of the virtual world. This doesn't mean that each and every room needs its own scene, but rather that each area

where rooms overlap requires two scenes, one for each room. Rooms that don't overlap with each other can then be grouped in the same scene. As a result, the number of scenes needed for overlapping rooms is equal to the maximum number of rooms that overlap each other in any given situation.

During development, care was taken to support applications with multiple simultaneously loaded scenes by not relying on any compile-time references between rooms or the player component, as these are not supported across scenes in Unity. Users of our implementation will have to be careful about cross-scene references as well if they want to use precomputed GI, which we consider a limitation of our current implementation. To allow per-room lightmap baking without separate scenes, we investigated Unity's render layers. They can be used to draw objects only on certain cameras and restrict physics ray casting to specific groups of objects. We tried to apply this restriction to the lightmap calculation process, but GI calculation is based on path *tracing*, a completely different implementation from physics ray casting and the camera drawing logic, without support for the render layer functionality.

A notable consequence of restricting lightmap baking to a single scene is that no lights from other scenes will contribute to the GI calculated for that scene. This means that a corridor between two overlapping rooms might need its lightmaps to be baked together with both rooms first, to prevent seams at the entrances of rooms, where light might shine from the rooms' interiors into the corridor. Afterwards, lightmaps for both rooms will have to be baked separately, overriding the lightmaps previously generated when baking all scenes at the same time. As a result, lights from inside a room can affect GI in a corridor through entrances, but rays from lights in a corridor will not affect GI inside an overlapping room. Following this procedure might require an additional scene to accommodate the geometry outside overlapping rooms.

4 Conclusion

In this paper, we described problems and challenges related to the implementation of impossible spaces in VR games. We presented our approaches that is available as an open source plugin for Unity and was already used for two different projects: a scientific study [7] and a commercial VR game [8]. The experiences of working with the implementation on this game are documented by Paulmann et al. [5].

References

1. Bozgeyikli, E., Raij, A., Katkoori, S., Dubey, R.: Point & teleport locomotion technique for virtual reality. In: Proceedings of ACM Symposium on Computer-Human Interaction in Play (CHI Play). pp. 205–216 (2016)

[7] https://www.xrdrn.org/2020/12/dead-science-orientation-and-minimaps-in-small-spaces/

[8] https://www.youtube.com/watch?v=eBXya_LiwFY/

2. Langbehn, E., Steinicke, F.: Space walk: a combination of subtle redirected walking techniques integrated with gameplay and narration. In: ACM SIGGRAPH 2019 Emerging Technologies, pp. 1–2 (2019)
3. Murray, T., Vigentini, L.: A study protocol to research and improve presence and vection in vr with a non-euclidean approach. In: 2019 IEEE International Conference on Engineering, Technology and Education (TALE). pp. 1–5 (2019)
4. Möller, H.: Antichamber: A strange world, https://hendrik.fam-moe.de/wp-content/uploads/2019/02/Antichamber_Analysis.pdf
5. Paulmann, H., Mayer, T., Barnes, M., Briddigkeit, D., Steinicke, F., Langbehn, E.: Combining natural techniques to achieve seamless locomotion in consumer vr spaces. In: 2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW). pp. 383–384. IEEE (2021)
6. Pixels, T.: *Unseen Diplomacy*. Game [HTC Vive] (April 2016), triangular Pixels. Played November 2016.
7. Room, V.: *Tea for God*. Game [SteamVR] (October 2018), void Room. Played October 2018.
8. Slater, M.: Place illusion and plausibility can lead to realistic behaviour in immersive virtual environments. Philosophical Transactions of the Royal Society B: Biological Sciences **364**(1535), 3549–3557 (2009)
9. Suma, E.A., Bruder, G., Steinicke, F., Krum, D.M., Bolas, M.: A taxonomy for deploying redirection techniques in immersive virtual environments. 2012 IEEE Virtual Reality Workshops (VRW) pp. 43–46 (2012)
10. Suma, E.A., Clark, S., Krum, D.M., Finkelstein, S., Bolas, M., Wartell, Z.: Leveraging change blindness for redirection in virtual environments. 2011 IEEE Virtual Reality Conference pp. 159–166 (2011)
11. Suma, E.A., Lipps, Z., Finkelstein, S., Krum, D.M., Bolas, M.: Impossible spaces: Maximizing natural walking in virtual environments with self-overlapping architecture. IEEE Transactions on Visualization and Computer Graphics **18**, 555–564 (2012)
12. Usoh, M., Arthur, K., Whitton, M.C., Bastos, R., Steed, A., Slater, M., Brooks, Jr., F.P.: Walking > Walking-in-Place > Flying, in Virtual Environments. In: Proceedings of ACM SIGGRAPH. pp. 359–364 (1999)
13. Vasylevska, K., Kaufmann, H.: Influence of path complexity on spatial overlap perception in virtual environments. In: Proceedings of the 25th International Conference on Artificial Reality and Telexistence and 20th Eurographics Symposium on Virtual Environments. pp. 159–166. Eurographics Association (2015)
14. Vasylevska, K., Kaufmann, H.: Towards efficient spatial compression in self-overlapping virtual environments. In: Symposium on 3D User Interfaces (3DUI) (2017)
15. Vasylevska, K., Kaufmann, H., Bolas, M., Suma, E.A.: Flexible spaces: Dynamic layout generation for infinite walking in virtual environments. In: 3D User Interfaces (3DUI), 2013 IEEE Symposium on. pp. 39–42. IEEE (2013)
16. Wetzold, R.: An editor for impossible spaces in virtual reality, https://blog.wetzold.com/2020/07/02/an-editor-for-impossible-spaces-in-virtual-reality/
17. Wetzold, R.: Non-euclidean stencil portals in virtual reality, https://blog.wetzold.com/2020/01/04/non-euclidean-stencil-portals-in-virtual-reality/
18. Wetzold, R.: Stencils sorting geometry, https://blog.wetzold.com/2019/11/03/stencils-sorting-geometry/