Henrique d'Escragnolle-Taunay

A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
March 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Henrique d'Escragnolle-Taunay

A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Orientador: Prof. Alberto Barbosa Raposo

Rio de Janeiro
Março de 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Henrique d'Escragnolle-Taunay

A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Alberto Barbosa Raposo
Orientador
Departamento de Informática – PUC-Rio

Prof. Waldemar Celes
Departamento de Informática – PUC-Rio

Prof. Marcelo Gattass
Departamento de Informática – PUC-Rio

Prof. Marcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, March 4^{th}, 2016.

Henrique d'Escragnolle-Taunay

BSc. in Information Systems at PUC-Rio - 2011. Worked at: TecGraf, developing virtual reality and scientific visualization systems; Olympya Software, developing a MMO game; Gapso, developing systems for business logistics; Bomnegocio.com/OLX, developing large scale micro-service oriented backend web services; and currently works at Microsoft, more specifically with search at the Bing team.

Bibliographic data

Taunay, Henrique d'Escragnolle

A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments / Henrique d'Escragnolle-Taunay; advisor: Alberto Barbosa Raposo. – 2016.

47f: il.; 30 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2016.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Computação Gráfica;. 3. Técnicas e Metodologias;. 4. Técnicas Interativas;. 5. Realismo e Gráficos Tridimensionais;. 6. Realidade Virtual.. I. Raposo, Alberto Barbosa. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To Carol, Bel, Malu and Peter.

Acknowledgments

I would like to express my gratitude to Professor Alberto Raposo, my research advisor for his support, guidance and useful critiques. I would also like to thank Pablo Elias, Vinicius Rodrigues and Rodrigo Braga, for their friendship and part in this work. I thank the PUC-Rio institution, especially the TecGraf lab, without them none of this would be possible. Finally, I would also like to thank Petrobras for funding our host project Siviep.

Abstract

Taunay, Henrique d'Escragnolle; Raposo, Alberto Barbosa(Advisor). **A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments**. Rio de Janeiro, 2016. 47p. MSc Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

With technological evolution, 3D virtual environments continuously increase in complexity; such is the case with multiscale environments, i.e., environments that contain groups of objects with extremely diverging levels of scale. Such scale variation makes it difficult to interactively navigate in this kind of environment since it demands repetitive and unintuitive adjustments in either velocity or scale, according to the objects that are close to the observer, in order to ensure a comfortable and stable navigation. Recent efforts have been developed working with heavy GPU based solutions that are not feasible depending on the complexity of the scene. We present a spatial partitioning heuristic for automatic adjustment of the 3D navigation speed in a multiscale virtual environment minimizing the workload and transferring it to the CPU, allowing the GPU to focus on rendering. Our proposal describes a geometric strategy during the preprocessing phase that allows us to estimate, in real-time phase, which is the shortest distance between the observer and the object nearest to him. From this unique information, we are capable to automatically adjusting the speed of navigation according to the characteristic scale of the region where the observer is. With the scene topological information obtained in a preprocessing phase, we are able to obtain, in real-time, the closest object and the visible objects, which allows us to propose two different heuristics for automatic navigation velocity. Finally, in order to verify the usability gain in the proposed approaches, user tests were conducted to evaluate the accuracy and efficiency of the navigation, and users' subjective satisfaction. Results were particularly significant for demonstrating accuracy gain in navigation while using the proposed approaches for both laymen and advanced users.

Keywords

Computer Graphics; Methodology and Techniques; Interaction techniques; Three-Dimensional Graphics and Realism; Virtual Reality.

Resumo

Taunay, Henrique d'Escragnolle; Raposo, Alberto Barbosa. **Uma heurística de partição espacial para o ajuste automático da velocidade de navegação 3D em ambientes de multiescala**. Rio de Janeiro, 2016. 47p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Com a evolução tecnológica, ambientes virtuais em 3D crescem continuamente em complexidade; este é o caso de ambientes multiescala, i.e., ambientes que contêm grupos de objetos com níveis de escala extremamente divergentes. Tal variação em escala dificulta a navegação interativa neste tipo de ambiente dado sua demanda repetitiva e não-intuitiva de ajustes em tanto velocidade quanto escala, levando em consideração os objetos que estão próximos ao observador, para garantir uma navegação estável e confortável. Esforços recentes tem sido desenvolvidos trabalhando com soluções fortemente baseadas na GPU que nem sempre podem ser viáveis dependendo da complexidade de uma cena. Nós apresentamos uma heurística de particionamento espacial para o ajuste automático de velocidade de navegação 3D em um ambiente multiescala virtual, minimizando o esforço computacional e transferindo este para a CPU, permitindo que a GPU possa focar na renderização. Nossa proposta descreve uma estratégia geométrica durante a fase de pré-processamento que nos permite estimar, em tempo real, qual é a menor distância entre o observador e o objeto mais próximo dele. A partir desta informação única, somos capazes de ajustar automáticamente a velocidade de navegação de acordo com a caraterística de escala da região na qual o observador se encontra. Com a informação topológica da cena obtida na fase de pré-processamento, somos capazes de responder, em tempo real, qual é o objeto mais próximo assim como o objeto visível mais próximo, que nos permite propor duas diferentes heurísticas de velocidade de navegação automática. Finalmente, com o objetivo de verificar o ganho de usabilidade alcançado com as abordagens propostas, foram realizados testes de usuário para avaliar a eficiência e precisão da navegação, assim como a satisfação subjetiva do usuário. Os resultados foram particularmente significantes ao demonstrar o ganho em precisão da navegação ao utilizar as abordagens propostas, tanto para usuários experientes quanto para leigos.

Palavras-chave

Computação Gráfica; Técnicas e Metodologias; Técnicas Interativas; Realismo e Gráficos Tridimensionais; Realidade Virtual.

Index

Figure List

Table List

We are what we repeatedly do; excellence, then,
is not an act but a habit.

Aristotle

1
Introduction

Freely navigating in a 3D virtual environment can prove to be problematic, even for the most experienced users [1], and possibly deal-breaking for laymen, especially when dealing with massive multiscale scenes. An example of this kind of scene, which was used in this work, is a real oil field, which varies in a scale of $1{:}10^7$ from the smallest object (an oil tube with a 15cm radius) to the largest (a seismic object with possibly kms of extension in all three dimensions). Some systems can tackle such scenarios more easily given their nature (e.g., examine focused applications, an exocentric interaction technique where the user can orbit and zoom in/out around a point of interest); however others that demand more navigation freedom (e.g., fly, an egocentric interaction technique) are more susceptible to user errors.

The problem of egocentric multiscale navigation has been tackled previously from mainly two distinct approaches: level of scale (LoS) based solutions, and automatic speed adjustment solutions. In LoS based solutions, the virtual environment surrounding the camera — or avatar — grows/shrinks according to user input [2] (i.e., a navigation with seven degrees of freedom (7DOF)); alternatively, the user can transit in and out from predefined discrete layers of scale [3]. The solution presented in this work follows the second approach, i.e. automatic speed adjustment, using the closest geometry position as input to heuristics that determine the optimal navigation speed at any given moment.

Examples of this last approach used an image-based environment representation named *cubemap* [4] [5]. Given the camera position, the cubemap is constructed from 6 rendering passes, each in a different direction in order to cover the whole environment. Targeting a more fluid navigation experience (i.e., without discrete scene scale layers or manual scale adjustment) with six degrees of freedom (6DOF), the cubemap technique is used to obtain an automatic speed adjustment for the scenario, which has proved to be an effective multiscale interaction technique solution.

However, this approach presents a different limitation: the render bottleneck. As virtual environment scenes grow in detail and complexity, despite the fast improvements in modern hardware, rendering six screens per frame is a GPU-intensive operation and can become unfeasible given the scenario. Following the motivation of eliminating such extra render steps, we propose a CPU based solution where the virtual environment's geometries are stored in a *k*-d tree [6]. This structure is used to obtain the nearest objects — visible as well as non-visible — allowing the application of a similar but revisited heuristic used in the cubemap solution.

The proposed solution presents a regression compared to the cubemap approach: it is only applicable to static scenes, given that rebuilding the *k*-d tree every frame can prove unfeasible. A proposed solution to this regression, working with parallel processing, has also been developed and will be introduced in this work as well.

The following chapter presents related work on multiscale navigation and *k*-d trees. In chapter 3 we show that such a solution matches the known cubemap features while successfully removing the render bottleneck without exhausting the CPU. The main divergences between both techniques will be exposed, and specific optimizations will be detailed as well. To back our usability claim, in chapter 4 we present results from a user-testing process involving participants with 3D navigation experience as well as laymen. Finally, in chapter 5, we will introduce an isolated and language agnostic tool that offers the optimal navigation speed heuristic feature as a service, while also allowing such solution to be scaled with ease.

2
Related Work

2.1
Automatic Navigation in 3D environments

The problem of automatic navigation in 3D environments was previously tackled by Mackinlay et al. [7]. They proposed a type of navigation which involved a user choice for a point of interest (POI). They addressed the difficulty of dealing with different speeds according to the POI, allowing you to move faster when you're far away from the object, and slower when you're close, making it possible for the user to carefully examine the desired object in a detailed manner. Although they developed a feasible solution, they assumed a discrete number of POI's in a scene and also limited the freedom of navigation considerably in their solution which is not suitable for all applications.

With the evolution of VR and 3D hardware and inevitably the growth of 3D scenes in size and complexity new kinds of user interaction problems have emerged. The term *multiscale environment* was forged by Perlin and Fox [8] to describe scenes in which a conventional navigation system is not sufficient to properly interact with a given environment (in their case 2D multiscale documents), and the option of adding the freedom to choose the scale with which the user would navigate was suggested.

Offering a 6DOF navigation is a considerable jump in interaction freedom and complexity compared to an examine interaction. Differently from Mackinlay's solution [7] in which a focus point was predetermined and therefore velocity control could be applied relative to such point, with 6DOF navigation it is not possible to determine exactly which is the scene object on which the user is currently focused at every given moment of the navigation. It is not difficult to see when navigating in a multiscale virtual environment that manually adjusting either velocity or scale can be unpractical and demand repetitive user inputs. Some techniques were developed with the goal of easing the navigation in these environments with a more universal approach. The most notorious one was showed by McCrae et al. [4], who used a render technique to fetch the nearest point to the observer at each frame, and used this point and its distance to choose an optimal velocity at that instant. Trindade and Raposo [5] extended the cubemap approach adding new features, such as determining automatically a pivot point when transitioning from free navigation to an examine navigation.

Another approach to tackling the 6DOF multiscale problem was the adaptive

navigation technique by Sanz [9], where not only the observers navigation speed is automatically adjusted, but the camera's rotation speed as well, in order to reduce jerkiness during the interaction. Sanz, instead of taking in as input the nearest object, worked with a combination of: a Time to Collision map, a grid where each cell represents the time to reach each rendered pixel given the user's current speed; and an Optical Flow map, a grid where each cell represents the amount of displacement (in screen space) between two consecutive frames. Both maps when applied to an custom algorithm can provide - what the author describes - as the perceived user's speed, which when compared to a hard coded optimal perceived speed (by configuration) serves as a reference for maximum translation and rotation variations between frames.

2.2 k-d Trees in Computer Graphics

Spatial data structures have been widely used in 3D computer graphic applications for a variety of purposes, and the *k*-d tree is one of the most popular choices in such a category. Among its advantages, the efficient point searching and closest neighbor calculation are very convenient for the usual geometric problems present in computer graphics.

Schaufler and Sturzlinger in 1996 presented an optimization technique for rendering complex virtual environment scenes, creating a cache of a scene's geometry stored in a *k*-d tree[10]. More recently Foley and Sugerman used the *k*-d tree to accelerate raytracing computed in the GPU when dealing with scenes with many objects in different scales[11], over the — until then — traditional grid acceleration structures. This solution was later improved by Horn [12]. However, previous solutions only worked with static scenes. A dynamic scene raytracing solution using a *k*-d tree in the GPU was later developed by Zhou et al. [13].

Other examples of the *k*-d tree usage in computer graphics are: real-time occlusion culling strategy for models that present large occluders, which replaced the traditional z-buffer approach with a *k*-d tree in which the scene's polygons were stored[14]; a real-time 3D pose estimation, exploring the spatial structure to obtain an efficient closest point computation used in comparison with predefined models [15]; and a panorama recognizing solution that manages to merge n 2D images into a single coherent image, utilizing the *k*-d tree for feature matching between multiple samples[16].

3
Automatic Speed Adjustment Heuristic

Our approach aims to be simpler and more efficient than the cubemap strategy. It is simpler because our objective is to obtain a good speed for each scene region, not necessarily the best, given that it is not necessary to treat the scene down to its lowest level of detail, and it is more efficient by making it possible to use heuristics to reduce CPU/GPU workload needed for an acceptable solution. In a complex multiscale scenario, such simplification is very important. Inspired by the work of McCrae's et al [4], we present a CPU based solution, in which most of the work is done in the preprocessing phase, leaving a minimal and efficient query to be performed in each frame, eliminating the need of extra render passes. This could easily allow an automatic speed strategy for scenes in which the rendering is the main bottleneck without exhausting the CPU.

The point of the work of McCrae's et al.[4] is to use its cubemap to fetch the nearest point to the observer in the scene. However, the nearest neighbor search is a well known problem that can be solved with the use of spatial structures, for example. In the following sub-sections, it is explained in detail how we developed a CPU based solution to an automatic speed navigation using the information from the result of a nearest neighbor search. It is shown how the scene is simplified to serve as input to a spatial structure, to reduce cost and to provide a good approximation of the optimal result. We present a math heuristic to determine the speed at each frame that allows the user to comfortably navigate in any position of the scenario. In addition, a technique to use the information stored in the spatial structure to fetch nearest visible point, which was used to improve our speed heuristic and implement features proposed by Trindade and Raposo[5].

We point out that the spatial structure based solution by itself currently does not deal with dynamic scenes, unlike the cited GPU solutions. Updating a *k*-d tree in real time is unfeasible given the complexity of the scenes we deal with, however, this problem can be solved with a hybrid solution. More on this topic will be covered in chapter 5, until then, we will discuss mainly the heuristic for static scenes, which is exactly the nature of the oil field scenes used as test case studies in this work through the SiVIEP system. More on SiVIEP will be presented on chapter 4.1.

3.1
The k-d Tree

The *k*-d tree [6] is a classic spatial structure that provides an efficient search of the nearest neighbor via a geometric approach. It is a space-partitioning data

structure for organizing points in a *k*-dimensional space represented by a binary tree in which every node is a *k*-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. An example of a two dimensional *k*-d tree is shown in Figure 3.1. In our scenario, exclusively 3 dimensions will be taken under consideration.

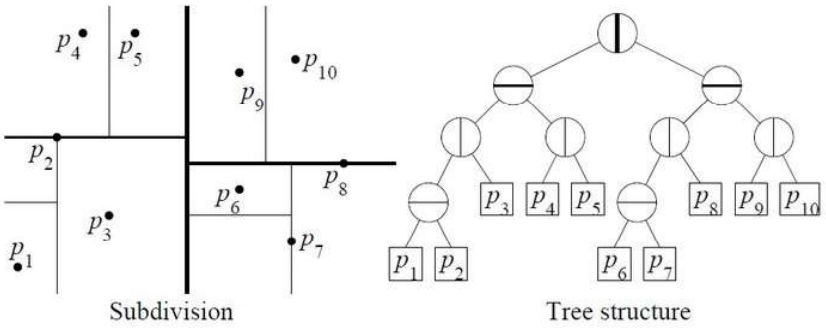

Figura 3.1: Example of a 2D *k*-d tree. The left image displays the points separated by the generated hyperplanes, and the right image presents the same structure in a binary tree view

The structure organization permits us to avoid big regions of the space by performing simple single number comparisons, such as distance to an orthogonal plane, speeding up operations while executing a geometric traversal. Algorithms like point search, nearest neighbor search and region search can be executed efficiently using a *k*-d tree. A balanced *k*-d tree performs the nearest neighbor search in O(log n).

A known limitation of this approach – compared to the cubemap solution – is that, since the tree nodes represent the vertices of a given scene's geometries, we can eventually face corner case scenarios where a relatively long and straight geometry (e.g. a tube) may not have any points along its body, and therefore no velocity adjustment would be applied along the surface between both edges. This problem was not tackled in this work.

3.2 Pre-Processing

The pre-processing phase of our approach aims to reduce the number of vertices to be considered when performing the query needed to calculate the instantaneous velocity. For that, our large scene is split into regular cubes of a given edge size named *cells*. These cells represent the basic unit of the velocity calculation

and a cell is considered filled if any vertex of any relevant object is located inside it, regardless of quantity. Knowing that, the whole scene is pre-processed to cluster all vertices into cells. The goal of this preprocessing phase is to reduce the space requirement and CPU realtime workload while using the *k*-d tree. One could decide to skip this phase and use all vertices without pre-processing, resulting in a more precise calculation, if the memory and processing resources are available. Also, its worth mentioning that this optimization process can eliminate part of the multi-scale nature of the scene, given that vertices in a smaller scale than the grid cell size will be discarded. However, this preprocessing phase makes the technique scalable and more efficient, since using raw vertices would not be applicable to all multiscale scenarios given their frequent huge sizes. Also, this provides flexibility for the navigation precision, being possible to set a larger cell size for a less precise velocity calculation and more efficient processing, and vice-versa.

The result of this phase is a sparse point cloud, orders of magnitude smaller than the original scene. The choice of the cell size has an important influence on how much the scene can be simplified and, as it will be seen later, how precise the navigation can be. The bigger the cell size, the more simplifications occur in the clustering phase. Figure 3.2 shows an example of this simplification.

In the example shown in Figure 3.2, using a cell with size 4 meters, the sample object, originally with more than 700,000 vertices, was simplified to only 1,400 cells. These cells are then stored in a *k*-d tree and this structure will be consulted in the real-time phase to properly calculate the navigation velocity.

3.3 The Real-Time Phase

At each frame, while a user is navigating, a nearest neighbor search is performed, having as parameter the cell where the user camera is located. The nearest neighbor search was first shown by Bentley [6] and improved by Friedman et al. [17], and is proved to cost O(log n) on the number of points present on the structure. Along with the simplification described in the last section, it is expected that such a search costs a small amount of time, even in real-time calculation. Having this nearest cell information in hand, along with the distance that can be easily calculated, we use a heuristic to calculate the speed the user can move at that instant.

The basic heuristic for an instant velocity is:

$$V = distance * cellSize \tag{3-1}$$

where *distance* means the number of cells of the *k*-d tree calculated from the nearest cell to the camera cell. The *cellSize* represents the length of a cell's edge in

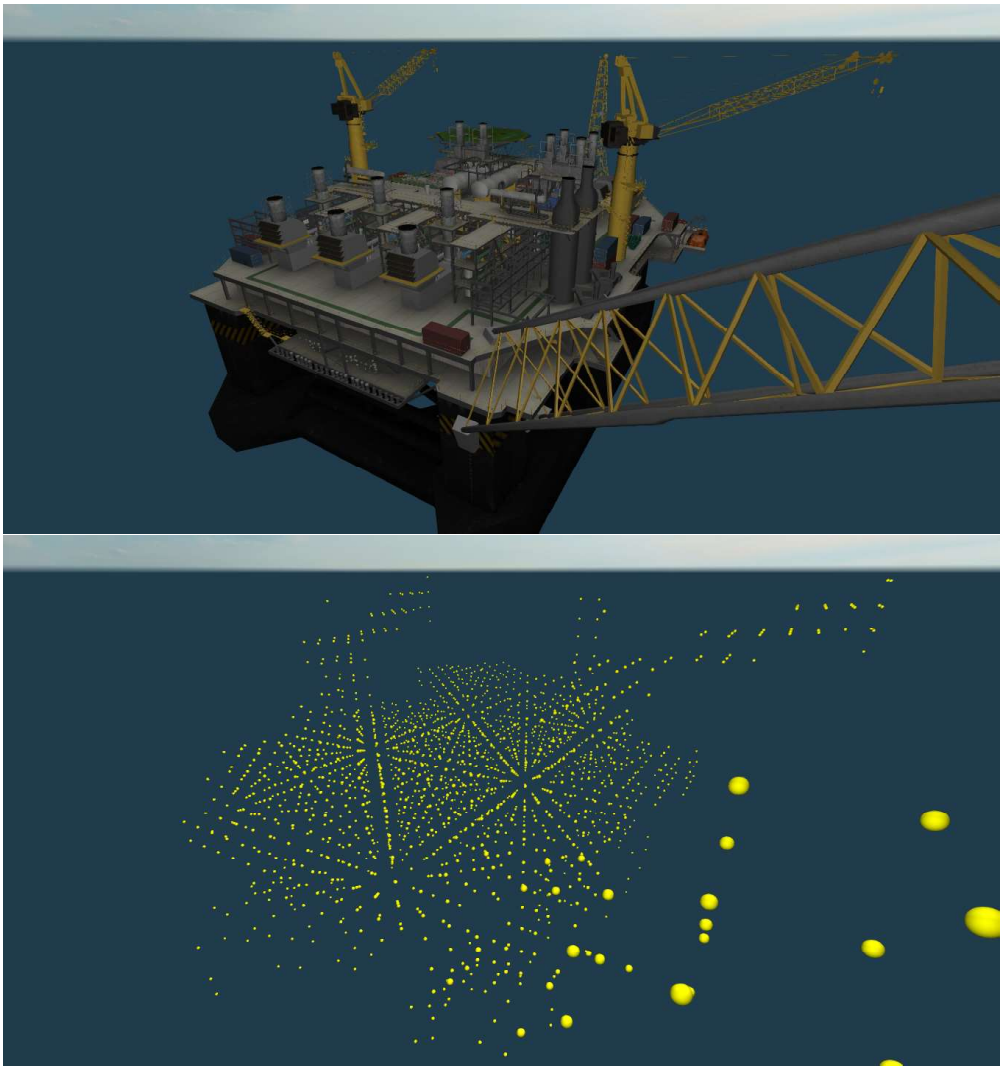

Figura 3.2: A complex object processed into cells

meters. So the instant velocity is (approximately) the distance to the observer per second.

However, this heuristic presented in equation 3-1 can cause abrupt changes in the current navigation velocity. For example, the larger the distance the more the current velocity increases, and hence the distance increments, and so forth. In order to reduce the probability of the user becoming disoriented, ideally the velocity variation should be smoothed. This can be achieved by limiting the acceleration/deceleration variations on the instant velocity in consecutive frames in a frame-rate independent fashion. We apply a smoothing function to the last velocity set, to calculate the current one. Our increment is limited by:

$$\Delta V = V_{t-1}^{A} \Delta t \tag{3-2}$$

where A is a constant potential increment factor. In other words, the current

velocity can accelerate at most $V^A_{t-1} - V_{t-1}$ per second. The final result of the velocity V in a instant t is:

$$V_t = V_{t-1} + (sgn(V_t - V_{t-1}) * \Delta V) \quad (3\text{-}3)$$

It's worth noting that this smoothing strategy can influence collisions with other scene objects, given that - depending on the scenario - the observer may not decelerate as necessary in order to avoid a theoretical collision. We also want to highlight that this strategy is in no way mandatory for the heuristic, during testing only a few scenarios demanded this adjustment, and therefore this strategy should be interpreted as a helper and optional feature to be enabled on demand.

The user instantaneous velocity is bounded between two values: V_{min} and V_{max}. V_{min} is chosen to be the cell size, while V_{max} is set according to the scale of the total dimension of the scene. These bounds are important in order to avoid both the user stopping or getting a speed so high that it gets uncontrollable. Therefore, for any t, we clamp the current velocity so $V_{min} \leq V_t \leq V_{max}$.

The influence of cell size can be perceived here. Since the minimum velocity is dependent on the cell size, it determines how precisely you can examine an object when you are close to it, or in other words, within the same cell. A good choice for a cell size depends on the smallest object in the scene that you would want to examine closely and carefully.

3.4 The Nearest Visible Search

One feature of Trindade and Raposo's[5] approach which remains to be solved in our strategy is the consideration of the "nearest visible point"for the automatic pivot point for exocentric navigation. Basically, it allows a smooth transition through an egocentric navigation to an exocentric navigation (examining an object) by setting a visible subject as the current point of interest. In their work, this information was obtained by a render strategy, more specifically, instead of obtaining the closest point in all 6 frames of the cubemap, only the closest point in the front frame was considered.

In our proposed implementation, we want to benefit from k-d tree's properties to make an efficient CPU based approach to obtain the same information. The k-d tree nodes entirely outside the view frustum could be discarded on the traversal for the search of the nearest neighbor, avoiding many unnecessary searches.

We present a strategy to perform a search that gives, as a result, the nearest neighbor within a view frustum. For simplification's sake, we present an algorithm considering the region only as the viewing frustum, but it can be extrapolated for any

region. For didactic reasons, we present the algorithm as recursive, but an iterative implementation is preferred to improve performance.

Consider, for each k-d tree node n, dim_n the dimension of the node that was split during its generation, and key_n the vertex used as key for that node. In our k-d tree, we consider that left nodes store keys that are smaller than the current node along its dimension. Consider also two arithmetic functions: *distance* that takes two points (or keys) as parameters and returns their euclidian distance; and *distToPlane* that returns the distance between a point p and an orthogonal plane, defined as follows:

$$distToPlane(n, p) = |p[dim_n] - key_n[dim_n]| \tag{3-4}$$

The algorithm is frustum aware and the model-view-projection matrix is used as an input. Previously, an axis aligned bounding box of the frustum geometry in world coordinates has been calculated, shown below as $frustum_{min}$ and $frustum_{max}$. The *isVisible* function, considering the input frustum, is assumed implemented, by projecting a point into clipping space, and checking if it belongs within the borders of the canonical cube.

In a regular multiscale scenario, the view frustum tends to be much smaller than the whole scene. Therefore, many branches of the k-d tree are readily ignored and, besides the additional plane-against-frustum tests, the search tends to be faster than the global nearest neighbor calculation on average. The result of this search can be used as a pivot point on exocentric navigation, similarly to Trindade and Raposo's solution [5].

3.5 Improving the Heuristic

A common issue with defining the current navigation velocity based on the nearest world point occurs when leaving a near object while facing a different distant object towards which the user wishes to navigate. Although the target object is relatively far, the previous object that is still near the camera — despite not being visible — limits the acceleration, resulting in a frustrating feeling of being pulled back.

Our proposed solution to this problem involves taking advantage of the result of the nearest visible search to extend the heuristic presented in section 3.3. Let p_{camera}, p_{global} and $p_{visible}$ be the camera position, the position of the nearest neighbor to the camera, and the nearest visible neighbor to the camera, respectively. Thus, we define two vectors (normalized):

Algoritmo 3.1 Nearest Visible Algorithm

$n \leftarrow root$
$nearest \leftarrow \infty$

function $nearestVisible(\ n, nearest, p\ )$
  if $key_n[dim] < frustum_{min}[dim_n]$ and $n.right \neq null$ **then**
    return $nearestVisible(\ n.right, nearest, p\ )$
  end if
  if $key_n[dim_n] > frustum_{max}[dim_n]$ and $n.left \neq null$ **then**
    return $nearestVisible(\ n.left, nearest, p\ )$
  end if

  $resultNode \leftarrow null$
  if $isVisible(key_n)$ and $distance(p, key_n) < nearest$ **then**
    $nearest \leftarrow distance(\ p, key_n\ )$
    $resultNode \leftarrow n$
  end if
  if $n.left \neq null$ and $distToPlane(n.left, p) < nearest$ **then**
    $tempNode \leftarrow nearestVisible(\ n.left, nearest, p\ )$
    if $tempNode \neq null$ **then**
      $resultNode \leftarrow tempNode$
    end if
  end if
  if $n.right \neq null$ and $distToPlane(n.right, p) < nearest$ **then**
    $tempNode \leftarrow nearestVisible(\ n.right, nearest, p\ )$
    if $tempNode \neq null$ **then**
      $resultNode \leftarrow tempNode$
    end if
  end if

  return $resultNode$
end function

$$\begin{aligned} \vec{v}_{global} &= p_{global} - p_{camera} \\ \vec{v}_{visible} &= p_{visible} - p_{camera} \end{aligned} \tag{3-5}$$

So we replace the calculus of V on the equation 3-1 for:

$$V = (distance * cellSize) * (1 + \frac{1 - \vec{v}_{global} \cdot \vec{v}_{visible}}{2}) \tag{3-6}$$

The sequence of the heuristic logic follows equally. The desirable result is to give priority to visible points when deciding the base velocity. When the nearest visible and nearest global are in completely opposite directions in relation to the viewer, the result velocity is doubled. If $p_{global} = p_{visible}$ then it behaves exactly as in the basic heuristic.

An undesired limitation of this improvement occurs when visible objects located near the border of the screen maintain the frustrating feeling of not accelerating accordingly towards the distant object on which the camera is centered. Assuming that the user will always center the camera in the direction on which he/she wishes to navigate, we improved once more the heuristic to only consider objects located relatively in the center of the camera view.

This proposed improvement is achieved by deliberately narrowing the viewing frustum by reducing the perspective *fovy*. This would consider only objects that are shown in the center of the visible area, as shown in Figure 3.3.

To recap, in the table 3.1 we present the list of all of the configuration values accepted by the heuristic presented in this work.

Name	**Description**	**Mandatory**
Cell Size	Grid cell size, and also input as min velocity (even if grid optimization is disabled)	Yes
A	Speed Increment Factor	No
V_{max}	Maximum speed limit	No
Visible Fovy	Field of view to be taken into account when performing the nearest visible search	No

Tabela 3.1: Configuration values

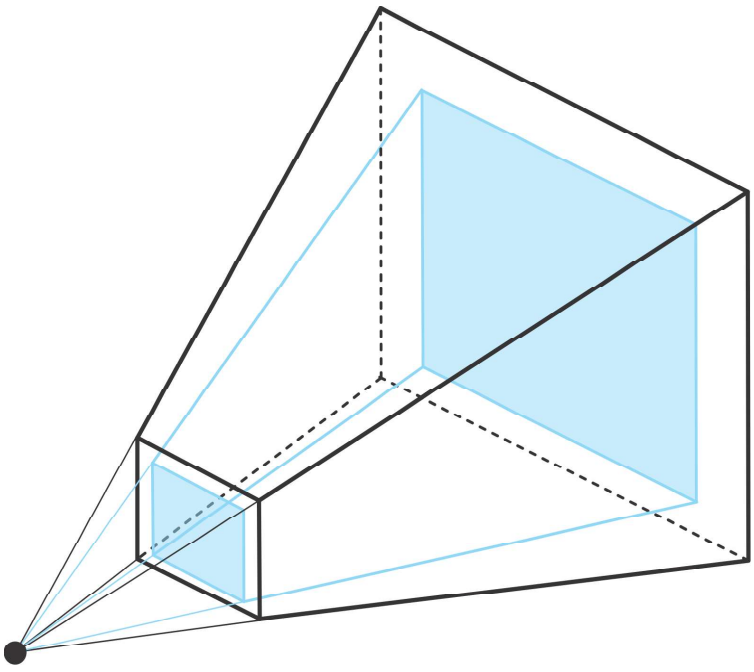

Figura 3.3: Example of how the *fovy* of the viewing frustum was reduced while searching for the nearest visible point. The black frustum represents the area being rendered, while the blue one represents the search area.

3.6 Performance

In this section we present the impact of the proposed strategies on application performance. Our test scenario has a total of 8.9 million vertices, 5.9 million primitives and 1250 unique objects. The specifications of the machine which ran the tests are: Core i7-920 (2.67 GHz) processor, 6 GB RAM memory, GeForce GTX 460 graphics card.

The performance test was executed by running a predefined camera path, trying to cover scene areas with different CPU and GPU demands. The same path was run using three situations: not using any automatic adjustment, using the basic heuristic for automatic speed adjustment (section 3.3), and using the nearest visible heuristic for automatic speed adjustment (section 3.5), named A, B and C, respectively. Table 3.2 shows general performance results, and Figure 3.4 shows the measurements per seconds in a graph.

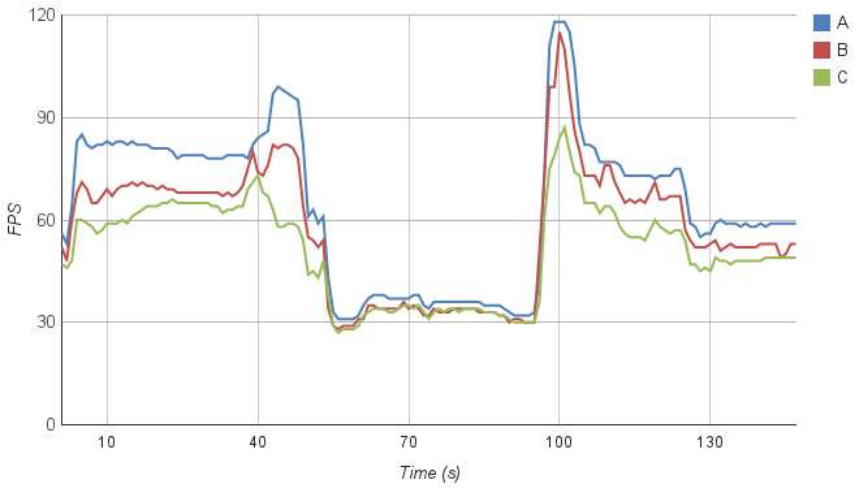

Figura 3.4: Graph of FPS measurement per time

FPS count	A	B	C
Average	64.38	57.10	50.97
Minimum	31	28	27
Maximum	118	115	87

Tabela 3.2: General results of performance for each strategy

As we can notice by Table 3.2, the impact is roughly a fixed rate (11%) for the B scenario on average, considering the scenario A as baseline. A similar cost rate is perceived between B strategy and C strategy (11%). The graph confirms the proximity, except in rare cases. It is also worth highlighting that the worst frame-rates in the B and C scenarios were nearly identical.

4
User Tests

To evaluate the usability of the proposed techniques, a batch of user tests were conducted. The techniques present the goal of assisting the users in the task of exploring a multiscale virtual environment with a more fluid, comfortable and intuitive experience. Therefore, we chose to perform tests comparing navigation experiences with and without such improvements. Among the aspects of navigation to be analyzed, we were mainly interested in the precision, duration and overall user satisfaction.

4.1
Test Environment

The tests were performed using the Siviep viewer, a project under development by TecGraf in cooperation with Petrobras. Siviep supports a comprehensive visualization of several types of models comprising an oil exploration and production enterprise. For example, it is possible to examine the oil extraction process starting at the reservoirs, passing through the wells, the water and gas pumps, up to the ducts that arrive at the oil platform, also included, not to mention seismic and terrain data as well. All of this is in a single 3D scene (see Figure 4.1).

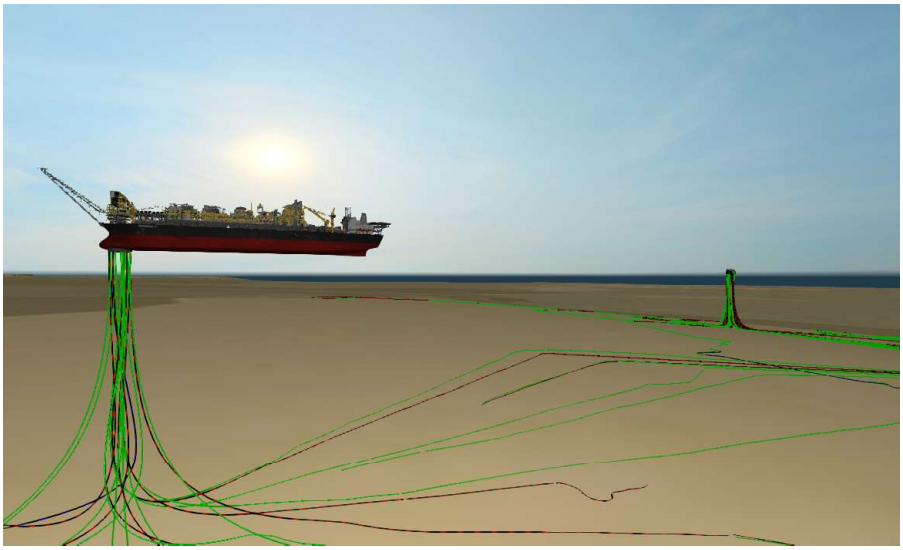

Figura 4.1: Example screenshot of Siviep

Such a scenario is relevant for the test because of the multiscale nature of the different types of models that can be inspected simultaneously in a single interactive scene.

4.2 User Profiles

The tests were conducted with a group of 24 subjects. Twelve of them already had previous experiences with 3D navigation (e.g., 3D modeling applications, video games, other 3D viewers), and the remaining twelve had little or no experience. We will refer to these subgroups as the *experienced* group and the *laymen* group respectively. The individuals in the experienced group are herein called E1 to E12, while the test users in the laymen group are identified as L1 to L12.

The ages of the subjects varied from 22 to 55, all of them were familiar with the input devices used in the experiment (keyboard and mouse), and none of them had any previous contact with the application used in the tests.

4.3 Test Design

The proposed test consists of the user navigating through a predefined path guided by a sequence of rings (Figure 4.2). Only one ring is visible at each time, and the user is instructed to attempt to navigate through such ring. Once surpassing the ring — be it successfully through its bounds or unsuccessfully outside — it will immediately disappear and the next ring will appear, and so on until the end of the test. In the case of the next ring in the course escaping the current view of the camera, an arrow indicating the direction of such ring is displayed on the screen to help the user.

The ring sequence forms a course covering most of the selected scene's elements, which is a convenient path for evaluating the automatic speed adjustment. While the two closest rings — both located inside an oil platform — present a distance of approximately 15m between them, the most distant pair of rings — located between the offshore enterprises and the continent's coast — have more than 80km separating them. The course can also be viewed as three separate sub-courses connected between themselves: inside platform navigation; between platform navigation; and offshore navigation, each with a particular scale, presenting average distances between rings at 50m, 1500m, and 40km respectively. And yet, the course was engineered to allow a navigation experience with similar time intervals necessary to advance through any pair of rings.

The navigation itself is performed from a first person perspective with 6DOF and using mouse and keyboard as input devices. The mouse serves as an interface to determine the direction in which the user is looking. During manual navigation, the mouse scroll wheel is used to determine the navigation velocity. The keyboard serves as an interface to determine the translation movement of the camera, always relative to the direction in which the camera is facing.

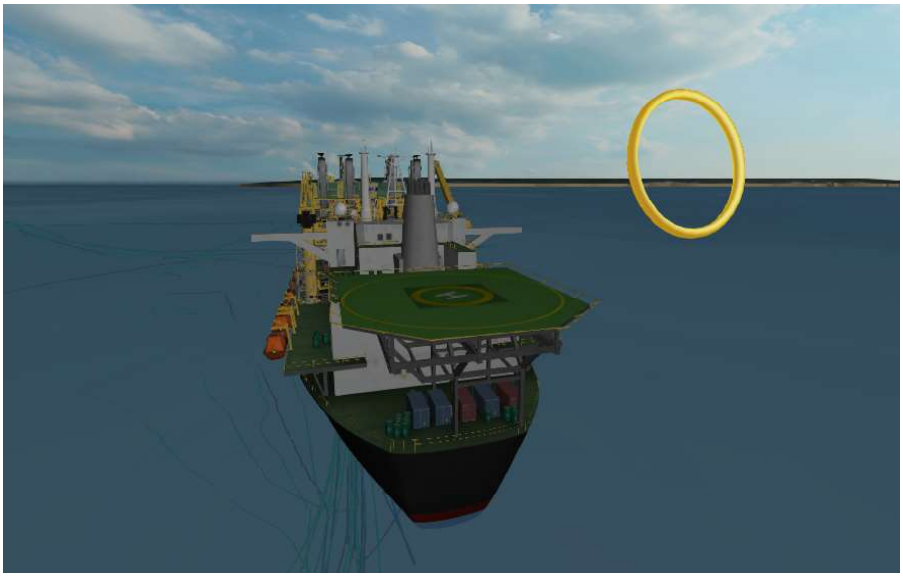

Figura 4.2: Ring in test

Each user was asked to perform three interactions on the same pathway by varying the velocity adjustment policy of each interaction: interaction A works with a fully manual speed adjustment system; B with a nearest-point automatic speed adjustment heuristic; and C with a hybrid nearest-point and nearest-visible-point speed adjustment heuristic, as seen in section 3.5. In interaction A the user was also offered feedback of the current velocity on the GUI to assist the navigation, while the B and C interactions offered no such feedback with the goal of making the speed adjustment as natural as possible. As far as the subjects were concerned, there were no apparent distinctions presented between interactions B and C.

This multiple-condition *within-subjects* test approach used the *counterbalancing* technique with a *Latin Square* order[18] to compensate the learning between interactions. An advantage of the within-subjects design is that there is less variance due to participant disposition, given that a participant who is predisposed to be meticulous (or reckless) will be likely to exhibit such behavior consistently across all interactions, and therefore the variability in measurements is more likely to be attributed to differences between interactions than differences between participants.

Each user was introduced to the system and the procedure identically, followed by the explanation of the current interaction based on the order of the given test, where only the information relevant for each type of interaction was informed incrementally. We purposely chose not to conduct any training previous to testing, since our typical use-case involves laymen performing a quick navigation without being introduced to the system. After each interaction the user was given a *System Usability Scale* (SUS)[19] form to fill out (see appendix 9), producing, at the end of the test, three SUS results per user. Following the second interaction using

automatic-speed adjustment (be it *B* or *C*, depending on the order), users were also asked if they noticed any difference between both automatic experiences, and in such a case what were the differences.

During the testing process the following data was recorded for each interaction: the number of rings that the user was able to successfully navigate through; the time taken to navigate between each pair of rings; the total distance covered between each pair of rings; the current state of the velocity of each frame step during the entire interaction; the actual velocity of each frame step, obtained by dividing the distance covered by the duration of such frame; and lastly the quantity of discrete inputs the user made during the interaction. A discrete input is defined by any time the user presses and releases a key from the keyboard, or when he/she starts and finishes a mouse wheel movement. Camera orientation defined by the mouse cursor is not recorded. To be clear, the reason to record both the velocity state and actual velocity is justified by the fact that each can offer distinct relevant data, e.g. the velocity state of each frame offers us a graph of exactly how the user altered his velocity manually or how the system adjusted it for him, while the actual velocity offers us information of when - and during which speed moments - the user stopped translating.

4.4 Test Results

In order to analyze the test results objectively, it is important that we interpret the data of both user groups — laymen and experienced users — separately, since our solution may affect each category differently. Users in the laymen category have difficulties dealing with the most trivial of multiscale navigations, and therefore our solution aims to allow an interaction that originally would simply not be possible, breaking the multiscale interaction barrier for non-experienced users. On the other hand, experienced users are already familiarized (and in some cases even comfortable) with manually adjusting navigation speed, and so our goal is focused on improving an already existing navigation experience, making it as fluid and intuitive as possible.

The normality prerequisite to perform a parametric significance test on our data was not met according the Shapiro Wilk test[20]. Therefore, the significance of the obtained data was tested using the Friedman test[21]. We assign this to the fact that, despite the groups being divided by their prior familiarity with 3D environments, other variables were not assessed, e.g., the person's ability or speed using mouse and keyboard. There were cases in which laymen performed similarly to some experts. In other cases, laymen were much less familiar with computer interaction than others in the same group and had difficulties with simple 3D

concepts, becoming clear outliers of the dataset.

4.4.1 Precision Analysis

The applied test consisted of a total of 30 rings through which the subjects should attempt to cross within their bounds as an evaluation of precision and control of the navigation system. Results showed an improvement in this criteria for both laymen and experienced users, as seen in Figure 4.3 when using automatic speed adjustment over the manual alternative. Curiously, while the experienced group managed to practically ace the test with both automatic solutions presenting a performance increase of nearly 16%, the laymen group felt more comfortable with the less volatile navigation technique *B* without the nearest visible object heuristic increment. This behavior can be understood by the difficulty that the unexperienced users had in dealing with the more abrupt changes in scale, and consequently in velocity. What one laymen would classify as an exaggerated jump in the acceleration in a short period of time, a more experienced user would consider as an essential volatility to avoid a frustrating experience of tediously waiting for his/her navigation velocity to change the desired value.

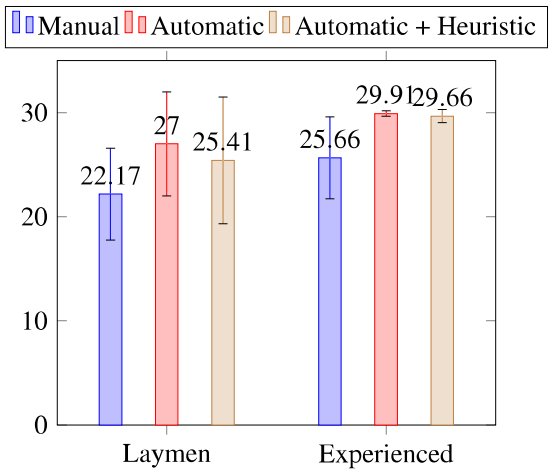

Figura 4.3: Average rings crossed successfully

According to the Friedman test, there was a significant statistical difference in precision measured with both groups (p=0.023 for laymen, p=0.001 for experts). Pairwise comparison between strategies showed that the most significant differences were between the manual speed strategy A and automatic speed adjustment strategies (Laymen AB: p=0.011, Experienced AB: p=0.001, and Experienced AC: p=0.004), showing that the automatic strategies improved the navigation precision. We did not find a significant difference between automatic strategies B and C in both groups. It should be highlighted that on both *B* and *C* interaction techniques

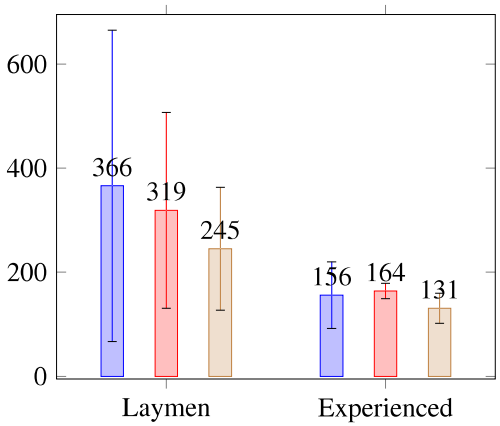

Figura 4.4: Average time to complete course in seconds

the laymen group managed to match or beat the experienced group accuracy level of manual navigation, i.e., with the aid of automatic speed adjustment a laymen user was able to perform similarly to an experienced navigator.

Regarding the completion time, no statistical relevance was observed with the laymen group using the Friedman test (p=0.205). Besides that, the laymen managed to achieve more precise results while taking considerably less time to complete the course on average — approximately 34% less comparing navigation technique *C* to *A* — as seen by observing Figure 4.4, which shows a tendency of improvement.

On the other hand, the Friedman test confirmed significant difference in the time measurements of the experienced group (p=0.009), despite the close average numbers. Pairwise comparison between strategies showed that the most significant difference was between the two automatic speed strategies B and C (p=0.02), showing that experienced users consistently improved their completion time using the visibility heuristic, without compromising their precision.

Figure 4.4 also shows us that experienced users already were able to navigate with near optimal speed control, as the variation between average course completion times between interaction techniques were negligible, while laymen were much more directly influenced by the different navigation approaches.

To better illustrate the learning curve of laymen when navigating in a 3D multiscale environment for the first time, Figure 4.5 shows the average course completion times separated by whether or not the manual navigation was the first test performed by the user. It is visible through the chart, as it was noticeable while applying the users tests, that the average user would struggle with the most simple transitions between the scene rings, implicitly frustrating the user and not allowing him/her to focus on the interaction as a whole as well as to get a better idea of what was expected from the test. On the other hand, when navigating manually after having experienced a more stable experience aided by the automatic speed

adjustment mechanism, the average user would still notice the limitations of manual velocity adjustment, but his/her familiarity with the test course as a whole flattened the manual adjustment learning curve. This behavior does not repeat itself when involving automatic navigation, as also seen in Figure 4.5. Automatic navigation completion times are hardly biased depending on whether the first navigation test was manual or not. We believe that even if the manual speed adjustment were to be preferred over the automatic solution by an experienced user, the automatic approach is more user friendly and more inclined to accelerate the learning process.

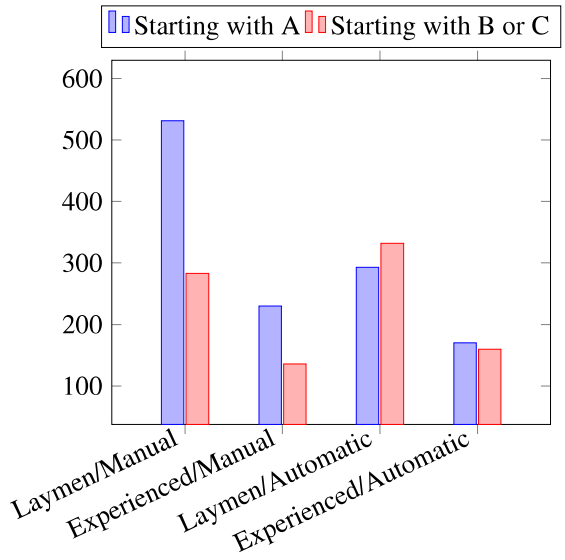

Figura 4.5: Average time to complete course, in seconds, depending on whether his/her first navigation test was manual or automatic

4.4.2 Input Analysis

By relieving the user of the responsibility of defining the navigation speed, the automatic speed adjustment technique demands considerably less user input without limiting movement freedom in any way. The results shown in Figure 4.6 reveal that tendency. The Friedman test showed no significant difference with the laymen group (p=0.174), but a statistical significance among the experienced group (p=0.005). Once again we notice that laymen are more comfortable with a less volatile velocity adjustment policy present in the *B* navigation scenario, while the experienced users presented approximately 50% improvement in both automatic approaches.

This drop in the demand for user input can be very useful depending on the device interface at hand. During testing, we worked with the mouse and keyboard devices, where both hands are used simultaneously offering a more flexible manipulation of the system. However, in immersive environments such as

caves, users usually have to work with a wand-like controller manipulated by a single hand, therefore overloading the quantity of inputs on a single device. Not having the worry about one of the interactions variables (speed) simplifies such a scenario.

A behavior observed during testing on manual velocity adjustment interactions was subjects showing difficulties in translating and adjusting their speed simultaneously. Some users, the majority of them laymen, would translate, stop, adjust their speed, and return to translating, resulting in a jerky experience. This issue is also solved by calculating the near-optimal velocity during navigation.

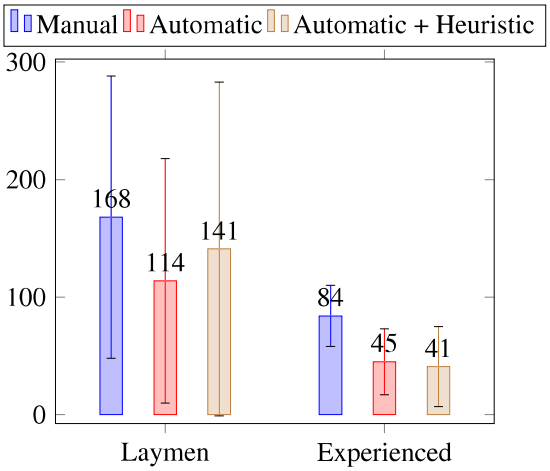

Figura 4.6: Average input count per interaction. A discrete input is defined as any time the user presses and releases a key from the keyboard, or when he/she starts and finishes a mouse wheel movement.

4.4.3 User Feedback

In order to evaluate the user experience of performing the navigation tests, we presented the subjects an SUS questionnaire after each interaction, resulting in the approval rates displayed in Figure 4.7. Both laymen and experienced groups showed improvements when interacting with the automatic velocity adjustment system, while laymen, once again, preferred the less volatile navigation technique *B*, and experienced users had near equal satisfaction with both *B* and *C* techniques. However, no significant difference was found according to the Friedman test (Laymen p=0.094, Experienced p=0.166). This disparity between SUS scores and the objective results from the study can be explained in part because SUS may not be the best questionnaire for a task-level evaluation [22].Another reason for this is that, in general, the users did not understand very well the differences between the automatic approaches.

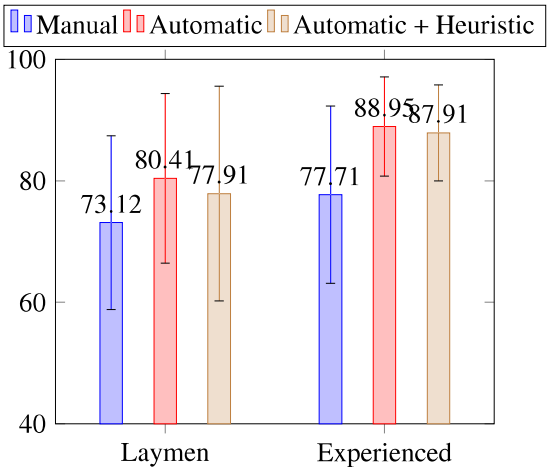

Figura 4.7: Average user SUS Scores

Despite experienced users presenting similar SUS ratings for both automatic techniques, and laymen even giving technique *B* a slight advantage over *C*, when asked if any difference was noticed between both types of automatic speed adjustment interactions, those who managed to notice the influence of the nearest visible point heuristic favored it over the only nearest-point alternative, as seen in Figure 4.8. While most users would offer less precise feedback such as *"C was faster"* or *"I felt more control with B"*, be it in favor or against the nearest visible point heuristic, three users were able to point out the exact improvements proposed, such as *"The interaction allowed me to leave objects faster, and decelerate faster as well when quickly approaching a smaller scale object"*.

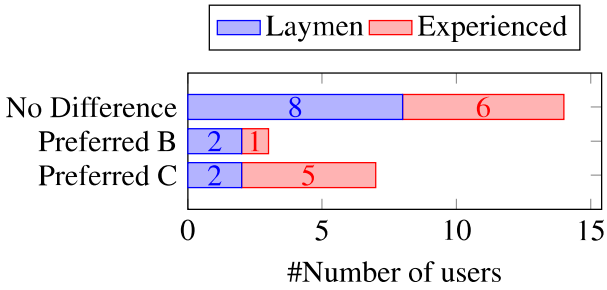

Figura 4.8: User feedback comparing interactions *B* and *C*

5
Automatic Speed Adjustment as a Service

After successfully completing the process of shifting the automatic speed adjustment calculation from the GPU to the CPU, we are no longer tied to a mandatory local solution, i.e. when the nearest point is obtained from the render process, it necessarily must be done on a machine where the entire 3D scene is being rendered, but this limitation does not apply when dealing with an abstract point grid only in the CPU.

A natural instinct would be to separate the multiscale calculation into its own thread, reducing completely the impact seen in the previous performance analysis 3.4, and allowing us to increase our number of points budget without worrying about impacting other CPU processes (e.g. scene graphs). However, this would still limit any practical solution to a single programming language or specific framework, without any necessity. The multiscale speed adjustment is a separate abstract representation of any given scene and has no need to be even in the same machine.

In order to offer a completely generic and agnostic solution to the multiscale navigation problem, we decided to create a service dedicated exclusively for solving it, allowing not only local but also remote access. In a nutshell, the service would allow any consumer to register points to it, populating a remote *k-d* tree, and later on querying which optimal velocity should be used with a given point in space and camera frustum.

This solution was inspired by a successful trend in large scale web applications, the microservice architecture[23], where systems are broken into several distributed services focused on offering a solution for a single problem. This mindset follows the popular Unix philosophy of ”do one thing, and do it well”, and not only is effective for ensuring modularity between components, but also conveniently fits well into the current cloud oriented direction the tech industry tending toward.

Therefore, as a conclusion to this research, we intend to deliver an agnostic, isolated and scalable service that offers an API for obtaining the optimal multiscale speed for any given 3D scene. In this chapter, we will dive into the technical specifics of the service solution - which we named Remote Multiscale Navigation System (RMNS) - present our performance results, and examine what other known problems this proposed solution helps us solve.

The RMNS is an open source initiative and all the source code, along with its documentation, tests and examples, is currently available at GitHub[24].

5.1 The Architecture

Our server solution, which we named Remote Multiscale Navigation System (RMNS), is responsible for:

- Receiving and registering information relevant for automatically determining the navigation speed of a given scene
- Answering which is the optimal velocity for navigation with the given inputs and previously registered information

The server is separated in two layers. On the top layer runs on Node.js's [25] javascript V8 engine, which os responsible for data checking, high-level logic and all HTTP communication. This top layer binds seamlessly with the a lower-level layer running on C++ process, which is responsible for all expensive geometric computation. All communication between both layers are asynchronous in order to avoid bottlenecks. See figure 5.1.

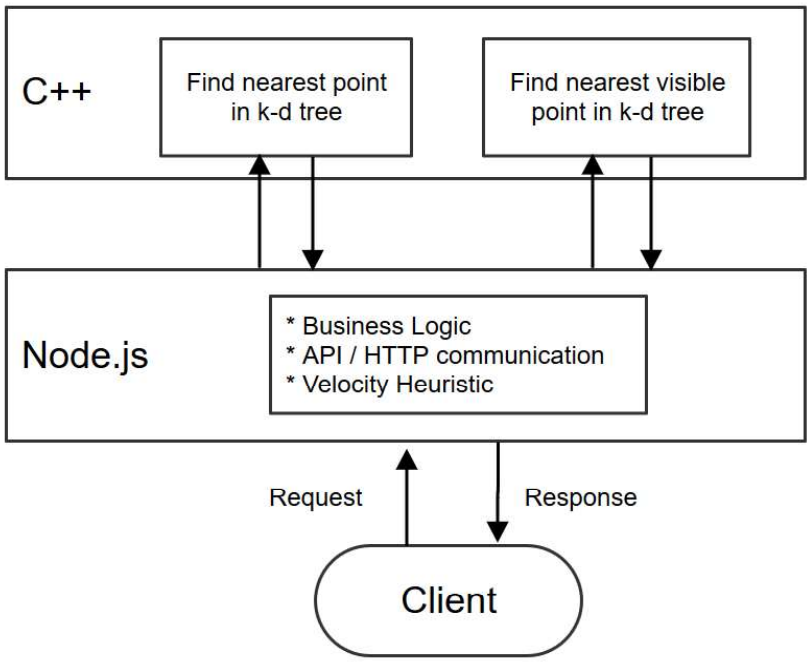

Figura 5.1: RMNS architecture

Custom settings such as cell grid size and fovy reduction - if any - are made by configuration and cannot be updated during execution.

A server can act following one of three different roles: stand-alone, master or slave. The former, as the name implies, is an independent approach where the entire system runs on a single Node.js process. The latter two are complementary

when working with a distributed solution. They allow, for example, that while one slave only deals with finding the nearest point, another can deal with just finding the nearest visible point, therefore parallelizing both efforts. In this scenario the master's role involves managing the communication between the slaves, as well as dealing with all business logic and the heuristic calculation.

The distributed approach leaves room for scaling scene complexities as well. Two different processes could be responsible for finding the nearest point in each half the scene, leaving the master to decide later on which one is closest. This improvement however could not be tackled during in time for this work and is postponed for future versions.

5.2 Dealing with Dynamic Objects

A downside from the CPU oriented nearest point solution is dynamic object support. While the GPU can seamlessly answer which is the nearest point in a given frame, without even having to be aware which objects are dynamic or not, rebuilding the *k-d* tree every frame is completely unfeasible for the CPU solution. This limitation did not pass unnoticed during the research and was given a lot of thought.

However, with a distributed system in place, we no longer need to be tied to the *k-d* tree exclusively. Imagine if while one process calculates the nearest point with the already known heuristics, a second process would calculate the nearest point taking into consideration only a relatively small subset of primitive objects (e.g. spheres and/or cubes). Figure 5.2 attempts to illustrate such a solution:

The advantages of using primitive objects are that they are cheaper to re-register on a frequent basis (data transfer wise) and do not demand rebuilding any spatial structure. Instead of transmitting several thousand points (or more) with a sphere we would need only a center and radius, for example. These primitive objects are stored in a list that on every request would be iterated linearly storing the nearest point found on each object, and returning only the closest one of all. Currently only spheres are supported as dynamic objects.

Since we are dealing with a list, and will be iterating it linearly, a reasonable suspicion could be raised over scalability issues. A benchmark was conducted with a mid-range server in order to measure this approaches performance, which proved that the bottleneck was not the linear loop as would be expected, which even with 1M spheres managed to maintain less than 40ms necessary taking into account both nearest global and visible points, but the limit of the body of the HTTP request package itself. In other words, the system can - within a acceptable time budget - register/update spheres in the scale of millions and on top of them calculate the

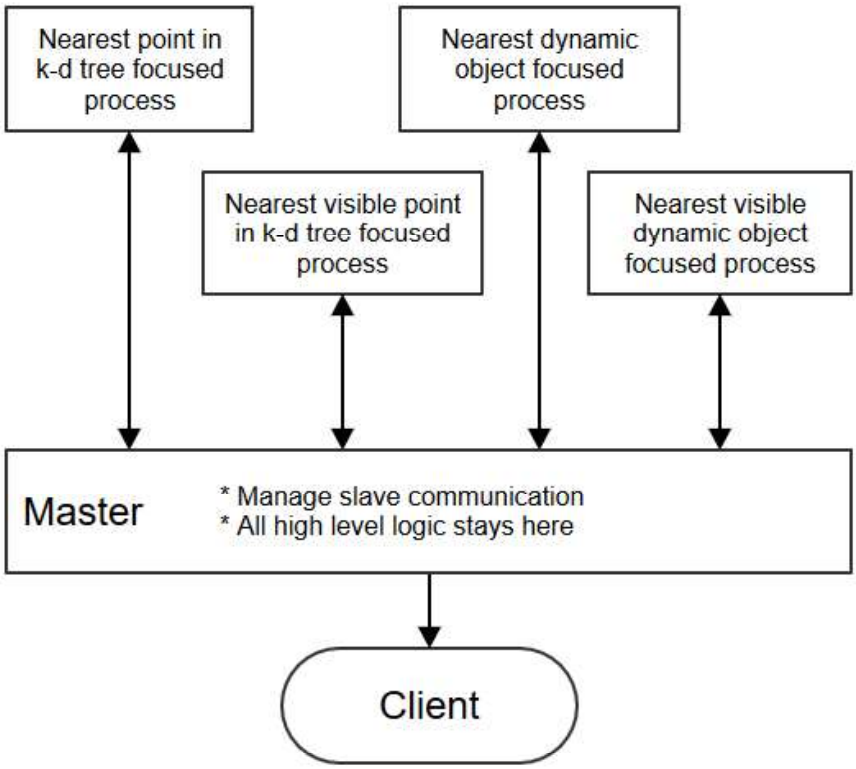

Figura 5.2: Distributed heuristic approach

optimal velocity heuristic, the issue is being able to update all sphere positions when an HTTP request can usually only transfer sphere data in the scale of tens of thousands. A workaround can be sending multiple sphere registration requests at a given frame, but in this case no test is needed to acknowledge that this approach would definitely not scale. Therefore, this dynamic-object solution is currently limited to a scale range of tens of thousands of objects.

It's worth noting that running RMNS in a distributed topology is not mandatory for dealing with scenes with dynamic objects. The stand alone mode also offers this feature, and if the scene's complexity and the machine's processing power allow it, navigation works seamlessly.

5.3 Consuming the API

All calls to the RMNS are by design asynchronous. This may lead to unfamiliar scenarios in computer graphics applications, such as a later call returning before a previous one. In a scenario where the current optimal speed is raising or decreasing in a constant ratio, responses that return from the server out of order may lead to a shaking and unstable navigation. A solution for this problem is to return in

every answer a timestamp, making the client responsible for verifying and eventually ignoring if any answers are already deprecated. The figure 5.3 exemplifies this situation.

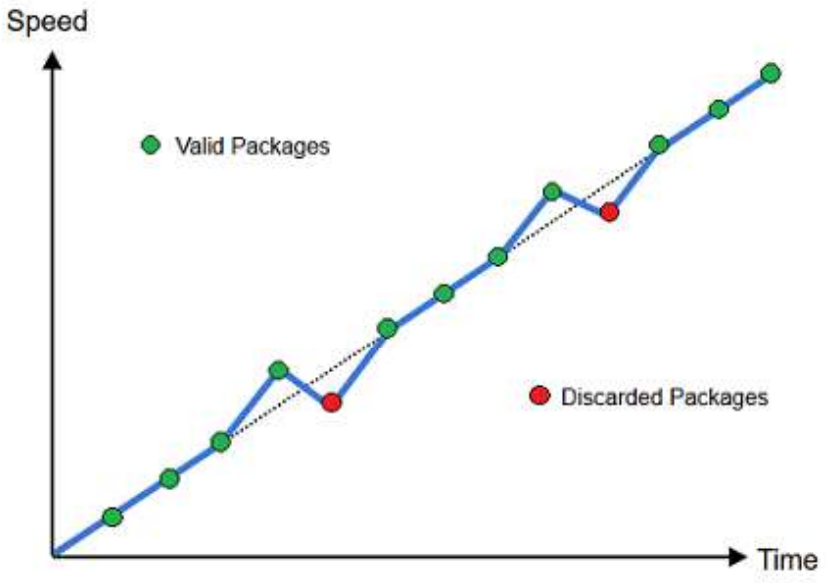

Figura 5.3: Navigation shaking for out of out responses

Currently there is support for point and basic geometry registration. The latter can, and should, be updated frequently during navigation, while the former must be registered previous to interaction, as rebuilding the point spatial structure in real-time is not supported. Any calls made to the point registration APIs during interaction will return an error response.

5.4 Solar System Experiment

As a demo test for the RMNS, we created a 3D scene representing the solar system, where each planet is represented by a sphere geometry, with their positions being updated on a frequent basis as dynamic objects, and the asteroid belt between Venus and Jupiter being represented by a 2M static size point cloud. The theme was chosen given the multiscale nature of the scene, that is also well known by the wide public. Figure 5.4 displays a screen-shot of the demo, which runs in the Unity3D engine (with C# code), proving the agnostic nature of the service. The demo can also be found at the project's web age[24], and is a good reference as an example on how to consume the RMNS.

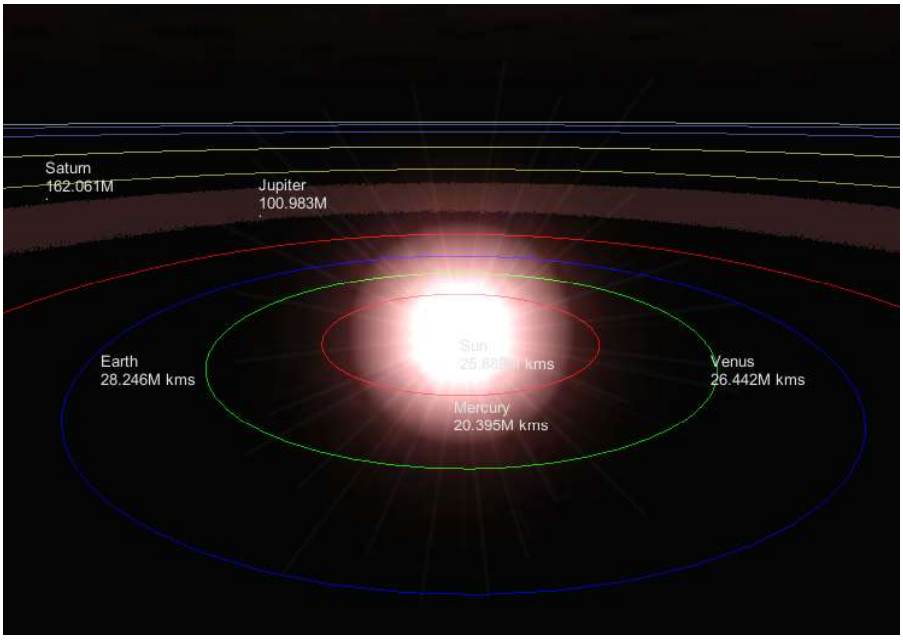

Figura 5.4: Solar system demo

5.5 Performance

The time taken for the RMNS to answer the optimal navigation speed is formed by the round-trip time plus the processing time on the server side. Taking into consideration the scenario where each process is run in parallel, as well as the fact that the optimal velocity can only be achieved when each of its answers are made available, we can conclude that the service is as fast as its slowest slave answer, plus the round-trip time. Figure 5.5 illustrates this behavior. In this example, processes A, B and C could be the nearest global point, the nearest visible point and the nearest visible and global sphere processes respectively, or any other combination of distributed processes as the system's users see fit. In fact, there could be more or less than three processes, since the service's generic architecture allows any number of distributed setups. Each distribution configuration should be fine-tuned depending on each scene's nature, targeting the minimization of the lengthiest process.

We have managed to obtain a 140ms answer time performance accessing from Rio de Janeiro, Brazil, a server running the RMNS in a data center in Texas, USA. From the 140ms total time, 20ms were from the round-trip and 120ms from the processing time bottleneck. This result was achieved with a single server. When distributing the service with a master server and two slaves performing the nearest point heuristic, a 90ms roundtrip time was achieved. In the distributed scenario, the bottleneck was the visible nearest point process, with an average 70ms processing

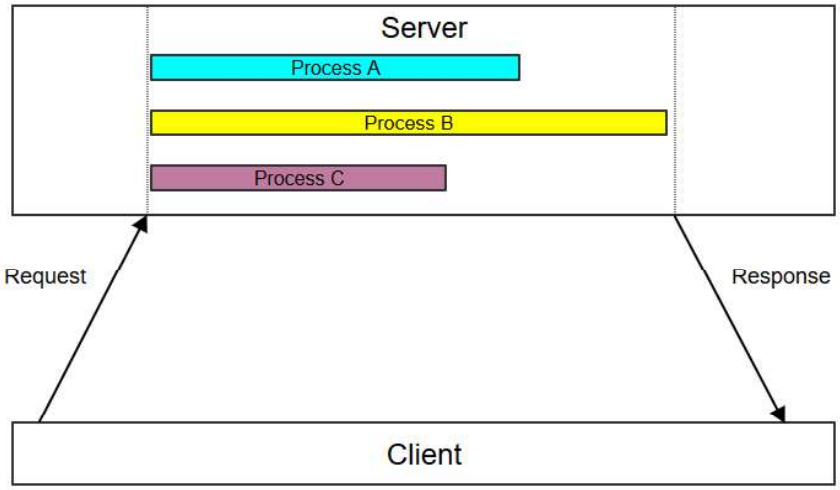

Figura 5.5: Round-trip path

time, that when added with the 20ms round-trip we reach the 90ms mark. Since RMNS still can't break a scene's *k*-d tree into separate processes this bottleneck currently cannot be any more parallelized. Now, despite that 140ms - or even 90ms - can be considered high-processing times in traditional synchronous computer graphic applications, the system's asynchronous nature does not affect in any way the main process, and therefore the roundtrip time is only relevant regarding how much it affects the user's navigation experience. The user tests presented in chapter 4 still were applied with the former local solution, so we are still not able to present formal usability results, but during the author's manual tests the delay did not appear to present any significant different - positive or negative - to the navigation experience. All performance tests were run working with an interval of approximately 10 requests for second, with a number of points in the scale of millions (4M to be exact) and spheres in the scale of thousands (5k to be exact), on top of mid-range virtual machines.

6
Conclusion

Multiscale navigation has proven to be a challenge to both experienced users and laymen, specifically regarding the task of defining the most suitable navigation speed for each moment during an interaction. Though it is still not a definite solution, since test subjects eventually still complained about the lack of fine tuning over the current velocity, there are indicators that removing this responsibility from the user improved the experience regarding control and overall satisfaction, and reduced the learning curve of the system. Laymen who previously were incapable of performing the most trivial interactions managed to complete our test course with the same precision as experienced users navigating with manual velocity adjustment. Experienced users averaged near perfect scores with half the inputs necessary for the manual technique while offering conclusive positive feedback on the SUS questionnaire.

The results achieved were similar to previous works [4] [5] but with more extensive testing. We also managed to evolve performance-wise, relocating the workload from the GPU to the CPU and consequently removing the need of rendering the same scene six times per frame, while at the same time reducing the overall processing demands of real-time interaction by working with a preprocessed spatial structure. This was achieved while maintaining most features available in previous similar solutions, with the exception of dealing with dynamic scenes indiscriminately (since the cost of updating the *k*-d tree in real-time is usually prohibitive). On the other hand, we were able to suggest a simple and complementary alternative by working with basic geometries that, while not being as generic as the previous GPU approach, can be proven useful depending on the scene being dealt with.

The proposed nearest-visible-point heuristic is a step towards improving the automatic-speed adjustment technique in a more universal solution. Although every alteration in the heuristic offers a trade-off, and due to the diversity present in multiscale scenarios, it is challenging to determine exactly what is the intention of the user. It is possible that eventually more advanced users could manually determine the heuristic improvements and adjustment variables more suitable for them.

The RMNS (Remote Multiscale Navigation System) is a initiative to make this solution available to the scientific community, while also opening the problem for parties interested in understanding or contributing towards the service. It also succeeds in isolating the problem from the main navigation system, providing an high level and language agnostic architecture interface, while also isolating the

automatic speed velocity computation by design.

For future works, we intend to explore the idea of breaking the scene into separate sub-scenes, in order to allow multiple processes to answer the nearest-point question in parallel. This feature will prove necessary the moment we start dealing with larger and more complex scenes where the grid strategy will not be able to reduce the total number of points enough to gain efficiency. We also intend to include support to more basic geometries - i.e. cubes and capsules - with the objective of offering more fine tuning and versatility when dealing with dynamic objects. Finally, on a different front, we plan to study the possibility of working with spatial structures that could allow reconstruction in real time, e.g. a more efficient variation of the k-d tree, or a BVH (bounding volume hierarchy). In the case of the BVH, it not only may prove helpful for dealing with dynamic objects, but it also has potential to be used as an alternative to the grid structure built during the pre-processing phase, with the goal of maintaning part of the multiscale nature of a given scene.

7
References

[1] George W. Fitzmaurice, Justin Matejka, Igor Mordatch, Azam Khan, and Gordon Kurtenbach. Safe 3D navigation. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 7–15. ACM, 2008.

[2] Xiaolong Zhang. Multiscale traveling: crossing the boundary between space and scale. *Virtual Reality*, 13(2):101–115, 2009.

[3] Regis Kopper, Tao Ni, Doug A. Bowman, and Marcio Pinho. Design and evaluation of navigation techniques for multiscale virtual environments. In *VR '06: Proceedings of the IEEE Virtual Reality Conference (VR 2006)*, page 24, Washington, DC, USA, 2006. IEEE Computer Society.

[4] James McCrae, Igor Mordatch, Michael Glueck, and Azam Khan. Multiscale 3d navigation. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 7–14, New York, NY, USA, 2009. ACM.

[5] DanielRibeiro Trindade and AlbertoBarbosa Raposo. Improving 3D navigation techniques in multiscale environments: a cubemap-based approach. *Multimedia Tools and Applications*, 73(2):939–959, 2014.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[7] Jock D. Mackinlay, Stuart K. Card, and George G. Robertson. Rapid controlled movement through a virtual 3d workspace. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 171–176, New York, NY, USA, 1990. ACM.

[8] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 57–64, New York, NY, USA, 1993. ACM.

[9] Ferran Argelaguet Sanz. Adaptive navigation for virtual environments. In *IEEE Symposium on 3D User Interfaces*, pages 91–94, 2016.

[10] Germot Schaufler and Wolfgang Sturzlinger. A three-dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):C227–C235, C471–C472, September 1996.

[11] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In Michael Meißner and Bengt-Olaf Schneider, editors, *Graphics Hardware*, pages 15–22, Los Angeles, California, 2005. Eurographics Association.

[12] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In Bruce Gooch and Peter-Pike J. Sloan, editors, *SI3D*, pages 167–174. ACM, 2007.

[13] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, December 2008.

[14] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 83–90. ACM SIGGRAPH, April 1997.

[15] D. Simon, M. Hebert, and T. Kanade. Real-time 3D pose estimation using a high-speed range sensor. In *CRA*, pages 2235–2241, 1994.

[16] Matthew Brown and David G. Lowe. Recognising panoramas. In *ICCV*, pages 1218–1227. IEEE Computer Society, 2003.

[17] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[18] I Scott MacKenzie. *Human-computer interaction: An empirical research perspective*. Newnes, 2012.

[19] John Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189:194, 1996.

[20] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, pages 591–611, 1965.

[21] Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[22] Jeff Sauro. *10 Things To Know About The System Usability Scale (SUS)*, October 2013. `https://www.measuringu.com/blog/10-things-SUS.php`.

[23] M Fowler and J Lewis. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014.

[24] Henrique d'Escragnolle Taunay. RMNS - Remote Multiscale Navigation System (https://github.com/htaunay/rmns), 2016.

[25] Ryan Dahl. Node. js: Evented i/o for v8 javascript. *URL: https://www. nodejs. org*, 2012.

8
Glossary

CPU Central Processing Unit. A computer's processor, more specifically the processing unit and control unit (CU), distinguishing these core elements from its external components such as main memory and I/O circuitry.

GPU Graphical Processing Unit. Has a different own internal architecture, which is parallel and customized for graphical applications.

Rendering Digital image generation by the physical simulation of light from data describing a scene.

SUS The System Usability Scale. A simple, ten-item attitude scale giving a global view of subjective assessments of usability

RMNS Remote Multiscale Navigation System. A stand-alone software as a service tool for defining the optimal speed for navigation given all the current scenes objects and a specific set a configurations.

9
Appendix: System Usability Scale (SUS) Form

System Usability Scale

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5