Real-Time Cinematic Tracking of Targets in Dynamic Environments

Category: Research

Abstract

Tracking a moving target in a 3D dynamic environment in a cinematic way remains a challenging problem through the need to simultaneously ensure a low computational cost, a good degree of reactivity to changes, and a high cinematic quality. In this paper, we draw on the idea of Motion-Predictive Control to propose an efficient real-time camera tracking technique which ensures these properties. Our approach relies on the predicted motion of a target to create and evaluate a very large number of camera motions using hardware ray casting. Our evaluation of camera motions includes a range of cinematic properties such as distance to target, visibility, collision, smoothness and jitter. Experiments are conducted to display the benefits of the approach with relation to prior work.

1 Introduction

The automated generation of cinematic camera motions in 3D virtual environments is a key problem for a number of computer graphics applications (computer games, automated generation of virtual tours, virtual storytelling). The first and foremost problem is to identify what are the intrinsic characteristics of a good camera motion. While film literature provides a thorough and in-depth analysis of what makes a viewpoint qualitative in terms of framing, angle to target, aesthetic composition, depth-of-field, lighting, the characterisation of camera motions has been far less addressed. This is pertained to specifics of real camera rigs (dollies, cranes) that reduce the set of motion possibilities, and the limited use of long camera sequences in movies (with the exception of steadicam sequence shots). In addition, the characteristics of camera motions in movies are strongly guided by the narrative intentions which need to be conveyed (*e.g.* rhythm, excitation, or soothing atmosphere).

In transposing the knowledge to the tracking of targets in virtual environments, one can however derive a number of desirable cinematic characteristics such as visibility (avoiding occlusion of the tracked target, and obviously collisions with the environment), smoothness (avoiding jerkiness in trajectories) and continuity (avoiding large changes in viewing angles and distances to target). In practice, these characteristics are often contradictory (avoiding a sudden occlusion requires a sudden acceleration, or a abrupt change in angle). Furthermore, the computational cost of evaluating visibility, collision, continuity and smoothness lowers the possibility of evaluating many alternative camera motions.

Existing work have either addressed the problem using global motion planning techniques typically based on precomputed roadmaps [5, 10, 11], or local planning techniques using ray casting [12] and shadow maps for efficient visibility computations [1, 2, 4]. While global motion planning techniques excel at ensuring visibility given their full prior knowledge of the scene, local planning techniques excel in handling strong dynamic changes in the environment. The main bottleneck of both approaches remains the limited capacity in evaluating expensive cinematic properties such as target visibility along a camera motion or in the local neighborhood of a camera.

Our approach builds on the idea of performing a mixed local+global approach by exploiting a finite-time horizon large enough to perform a global planning, yet efficient enough to react in real-time to sudden changes. This sliding window enables the real-time evaluation of thousands of camera motions by exploiting recent hardware raycasting techniques. As such, our approach draws inspiration from Motion Predictive control techniques [13] by optimizing a finite time-horizon, only implementing the current timeslot and then repeating the process on the following time slots.

A strong hypothesis we make is that the target object is controlled by the user through interactive inputs, hence its motions and actions can be predicted within a short time horizon H. Our system comprises 2 main stages, illustrated in Figure 1. In the first stage, we predict the motion of the target over a given time horizon H^i by using the target's current position (at time t) and the user inputs. We then select an ideal camera position at time $t+H^i$ and propose to define a *camera animation space* as a collection of smooth camera animations which link the current camera position (at time t), to the ideal camera location (at time $t+H^i$). In the second stage, we perform an evaluation of the quality of the camera animations in the animation space by relying on hardware raycasting techniques and select the best camera animation. In a way similar to motion-predictive control [13], we then apply part of the camera animation and re-start the process at a low frequency (4 Hz) or when a change in the user inputs is detected. Finally, to adapt the camera animation space to the scene configurations, we dynamically adapt a scaling factor on the animation space. As a whole, this process allows generating a continuous and smooth camera animation which enables the real-time tracking of a target object motions in fully dynamic and complex environments.

Our contributions are:

- the design of a camera animation space as a dedicated space in which to express a range of camera trajectories;
- an efficient evaluation technique using hardware ray casting;
- a motion predictive control approach that exploits the camera animation space to generate real-time cinematic camera motions.

2 Related Work

We narrow the scope of related work to real-time camera planning techniques. For a broader view of camera control techniques in computer graphics, we refer the reader to [3].

Global camera path planning

Global camera path-planning techniques build on well-known results from robotics such as probabilistic roadmaps, regular cell-decompositons or Delaunay triangulations. All have in common the prior computation of a roadmap as a graph where nodes represent regions of the configuration-free space (points, regular cells or other primitives), and edges represent collision-free links between the nodes. Niewenhuisen and Overmars exploited probabilistic roadmaps (PRM) to automatically perform queries withe the graph structure, linking given starting and ending configurations [10]. Heuristics were required to smooth the camera trajectories and avoid sudden changes in position and camera angles. Later, Oskam *et al.* [11] proposed a visibility-aware roadmap, by using sphere-sampling of the configuration-free space, and precomputing the combinatorial sphere-to-sphere visibility using stochastic ray-casting.

Lino *et al.* [7] exploited spatial partitioning techniques as dynamically evolving volumes around targets. Connectivity between volumes allowed to dynamically create a roadmap, through which camera paths were computed while accounting for visibility and viewpoint semantics along the path.

More recently, Jovane *et al.* exploited the 3D environments to create topology-aware camera roadmaps that would lower the roadamp complexity (compared to probablistic roadmaps) and enable the exploitation of different cinematic styles.

Yet, in all cases, the cost of precomputing the roadmap, and the difficulty in dynamically updating it to account for change in the 3D environments limits the practical applicability of such techniques to strongly dynamic environments, such as those met in computer games or storytelling applications.

Local camera planning

The other class of real-time camera planning techniques relies on a local knowledge of the environment. Mostly by sampling and evaluating the local neighborhood around the current camera location, such system are able to take decisions as to were to move at the next iteration, while evaluation classical cinematic properties such as visibility, smoothness and continuity. To address the computational issue of evaluation visilibity of targets, Halper *et al.* [4] exploited shadow maps to compute potential visible sets, coupled with a hierarchical solver. Normand and Christie exploited slanted rendering frustums to compose spatial and temporal visibility for two targets over a small temporal window (10 frames) [2]. Additional criteria were added in order to select the best move to perform at each frame, and to balance between camera smoothness and camera reactivity. Litteneker *et al.* [8] proposed a local planning technique based on an active contour algorithm.

Burg *et al.* [1] performed shadow map projections from the targets to the surface of the Toric manifold (a specific manifold space dedicated to camera control [6]). The visibility information provided by the shadow maps was then exploited to move the camera on the surface of the Toric manifold while ensuring secondary visual properties.

Recently, for the specific case of drone cinematography, Nageli *et al.* [9] built a non-linear model predictive contouring controller to jointly optimize 3D motion paths, the associated velocities and control inputs for a drone.

Our approach partly builds on the work of Nageli *et al.* , by borrowing the idea of a receding horizon process in which motion planning is performed for a large enough time horizon (few seconds), and the processed is repeated at a higher frequency to account for dynamic changes in the environment. Rather than addressing the problem using a non-linear solver, we propose in our paper to exploit the hardware raycasting capacities of recent graphics cards to efficiently detect collisions and occlusion and evaluate thousands of camera trajectories for each time slot.

3 Overview

Our system aims at tracking in real-time a target object traveling through a 3d animated scene, by generating a series of smooth cinematic-like camera motions.

In the following, we will present the construction of our camera animation space (Section 4), before detailing the evaluation of camera animations in this space (Section 5). Finally, we will show how we dynamically adapt and recompute our graph to fit the scene geometry and improve the results (Section 6).

4 Camera animation space

We propose the design of a *Camera Animation Space* as relative local frame, defined by an initial camera configuration $\mathbf{q}_{start}$ at time t and final camera configuration $\mathbf{q}_{goal}$ at time $t+H^i$ (see Figure 2). This local space defines all the possible camera animations that link $\mathbf{q}_{start}$ at time t to $\mathbf{q}_{goal}$ at time $t+H^i$. Our goal is to compute the optimal camera motion within this space considering a number of desired features on the trajectory (smoothness, collision and occlusion avoidance along the camera animation, ...).

H^i	Time horizon for iteration i (between times t_i and t_i+h)
$\mathbf{B}^i(t)$	Target behavior (predicted position) at time $t \in H_i$
$\mathbf{V}^i$	Set of preferred viewpoints at time t_i+h
$\mathbb{Q}^i$	Camera *animation space* for horizon H^i
M^i	Transform matrix of the camera animation space, for H^i
$\mathbf{q}^i_j(t)$	3D position in camera animation $\mathbf{q}^i_j \in \mathbb{Q}^i$, at time $t \in H^i$
$\mathbf{q}^i_{start}$	Starting camera position. $\mathbf{q}^i_j(t_i) = q^i_{start}$, $\forall\ (i,j)$
$\mathbf{q}^i_{goal}$	Goal camera position. $\mathbf{q}^i_j(t_i+h) = q^i_{goal} \in \mathbf{V}^i$, $\forall\ (i,j)$
$\dot{\mathbf{q}}(t)$	Tangent vector of a camera track at time t
$D^i_j(t)$	The camera view vector at time t
$(\mathbf{x},\mathbf{y})$	angle between two vectors $\mathbf{x}$ and $\mathbf{y}$

Table 1: Notations used in the paper

To this end, we propose to follow a 3-step process: (i) anticipate the target object's behavior (*i.e.* its next positions) within a given time horizon, (ii) choose a goal camera viewpoint from which to view the target at the end of the time horizon, and (iii) given this goal viewpoint, and the current one, build and evaluate the space of possible camera animations between them.

4.1 Anticipating the target behavior

We here make the strong assumption that we can anticipate, with good approximation, the next positions of the tracked target within a time horizon H^i. This is classical in character animation engines to select the best animation to play (*e.g.* motion matching). We consider H^i begins at time t_i and has a constant user-defined duration of h seconds. Moreover, we consider that the target behavior will be consistent over the whole horizon H^i. In our implementation, we consider the target as a rigid body (*e.g.* a capsule) with a current speed and acceleration, launch a physical simulation over H^i, then store all simulated positions of the rigid body over time. With this anticipation, we account for the scene geometry which might influence future user inputs, *e.g.* to make the target avoid collisions. We then refer to the anticipated positions as the target behavior, output in the form of a 3d animation curve $B^i(t)$ with $t \in H^i$ (see Figure 3). Note that one may use another technique to anticipate the target behavior, provided it can output a 3d animation curve $B^i(t)$ over time, it will not change the overall workflow of our camera system.

4.2 Selecting a goal viewpoint

We now make the second assumption that the user defines a set of viewpoints to portray the target object. By default, one might use a list of stereotypical viewpoints in movies such as 3-quarter front and back views, side views, or bird eye views. These viewpoints are sorted by order of preference (fixed by the user) in a priority queue $\mathbf{V}$. Each preferred viewpoint is defined as a 3d position in spherical coordinates (d,ϕ,θ), in the local frame of the target's configuration, where (ϕ,θ) defines the vertical and horizontal viewing angles, and d the viewing distance.

Given this set of viewpoints we propose, each time we update the target behavior, to select a good viewpoint where the camera should be at the end of the time horizon H^i, *i.e.* at time t_i+h. Considering all viewpoints are in $\mathbf{V}$, we pop viewpoints by order of priority. We propose to stop as soon as a viewpoint is promising enough, *i.e.* at time t_i+h neither the target will be occluded from this viewpoint, nor the camera will be in collision with the scene geometry. We then refer to this selected viewpoint as the goal viewpoint $\mathbf{q}_{goal}$.

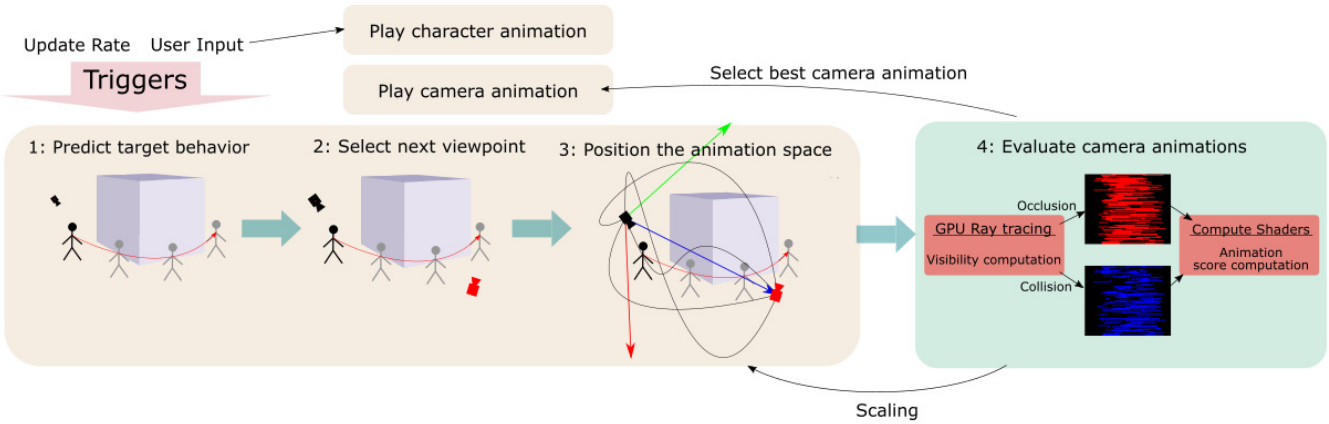


Figure 1: System overview: the orange box represents the CPU part of the system; the green box represent the GPU part of the system

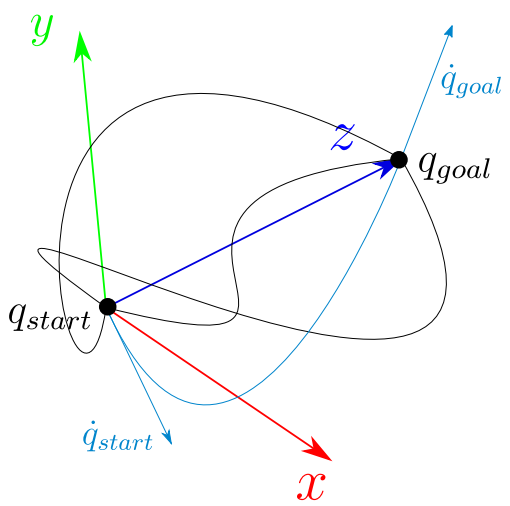


Figure 2: Representation of the *animation space* and its local transform

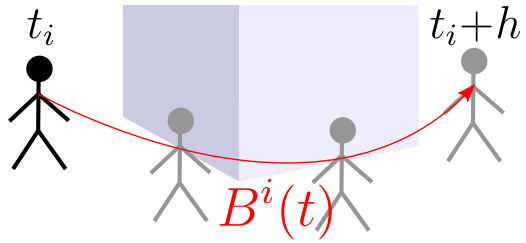


Figure 3: Representation of the target's behaviour curve at the ith iteration

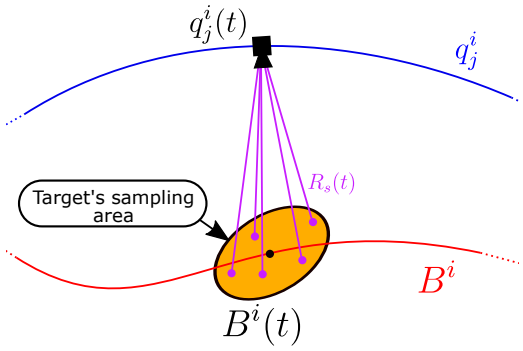


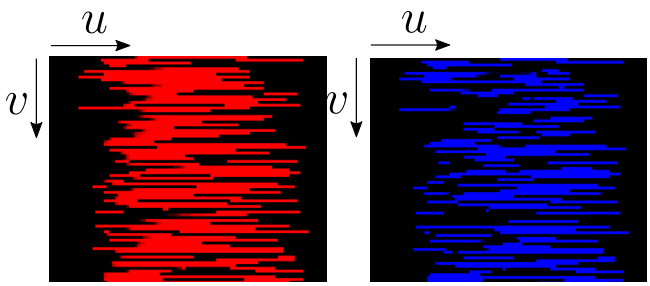
Figure 4: Ray launched from the camera toward the target's sampling area at time t

Knowing the current camera viewpoint $\mathbf{q}_{start}$ (at time t_i) and this goal viewpoint $\mathbf{q}_{goal}$ (at time $t_i + h$) provides a good basis to build camera animations that can follow the target behavior. We now need to categorize the full range of possible camera animations.

4.3 Sampling camera animations

Given the target behavior to track, we have computed two key viewpoints $\mathbf{q}_{start}$ and $\mathbf{q}_{goal}$, where the camera should be at start and end times of horizon H^i. We now propose to categorize the space of possible camera animations, between these two viewpoints. It is worth noting that this space is infinite, which makes it difficult to explore and evaluate. Our idea is to instead create a compact representation of this space, by sampling a large set of animation curves, with a good coverage of the range of animations. We will hereafter note this stochastic set of camera animations as $\mathbb{Q}^i$, and a sampled camera animation as $\mathbf{q}^i_j$, where j is the sample index.

Two requirements should be considered on this sampled space: (i) sampled camera animations should be as-smooth-as-possible, *i.e.* with low jerk, and (ii) the sampled animation space should enable to enforce continuity between successive horizons. To do so, we propose to encode each sampled camera animation as a cubic spline curve on all 3 camera position parameters, as they offer C^3 continuity between key-viewpoints. In practice, we make use of Hermite curves. They provide an easy means to sample the space of possible animations, by simply sampling a set of tangent vectors to the spline curve at start and end positions. Then, C^1 continuity between successive Hermite curve portions is commonly enforced by aligning both positions and tangents at connecting positions. We



(a) Occlusion (b) Collision

Figure 5: Example of a part of a Visibility data encoding texture; Black = The target is visible from the camera, Red = The target is occluded or partially occluded from the camera, Blue = The camera is inside the scene geometry

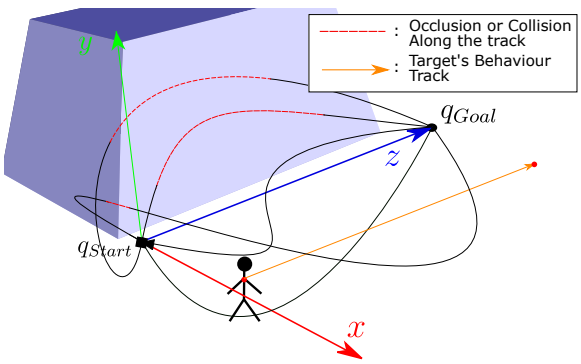


Figure 6: Representation of the positioned *animation space* that is asymmetrically collided by the scene geometry

hence propose to rely on the same idea.

In practice we propose, for each camera animation, to complement the starting and the goal camera positions $\mathbf{q}_{start}$ and $\mathbf{q}_{goal}$, by two tangents, *i.e.* the camera velocities $\dot{\mathbf{q}}_{start}$ and $\dot{\mathbf{q}}_{goal}$ (figure 2). To offer a good categorization of the whole *animation space* , we use a uniform sampling of these tangents in a sphere of radius r (in our tests, we used $r = 5$). The number of sampled animations is left as a user-defined parameter. Though, providing a sufficient categorization might require to sample several hundreds of animations (an evaluation of results for different values is provided in section 7.2).

Recomputing such a stochastic graph might be costly and/or lack stability over time. Our last proposition is then to precompute a graph of uniformly sampled camera animations, in an orthonormal coordinate system (as illustrated in figure 2). In this system, $\mathbf{q}_{start}$ and $\mathbf{q}_{goal}$ has coordinates $(0,0,0)$ and $(0,0,1)$ respectively. Then, for any horizon H^i, we apply a proper 4x4 transform matrix M^i to align the graph onto the computed viewpoints $\mathbf{q}^i_{start}$ and $\mathbf{q}^i_{goal}$. It is worth noting that in M^i the 3d translation, 3d rotation and the scaling on the z axis will lead this axis to match the vector $(\mathbf{q}^i_{goal} - \mathbf{q}^i_{start})$. Two parameters remain free: the scaling for the other two axes (x and y). As a first assumption we could use the same scaling as for z. However, we will further explain how to choose a better scaling in section 6, to take collisions and occlusions into consideration.

5 Evaluating Camera Animations

In the first stage, we have computed a set of camera animations $\mathbb{Q}^i$, that can portray the target objects' behavior within time horizon H^i. We now need to select one of these animations, as the one to apply to the camera. Our second stage is devoted to evaluating of the quality of all animations, and selecting the most promising one, in an efficient way. In the following, we will first detail our evaluation criteria, before focusing on how we perform this evaluation.

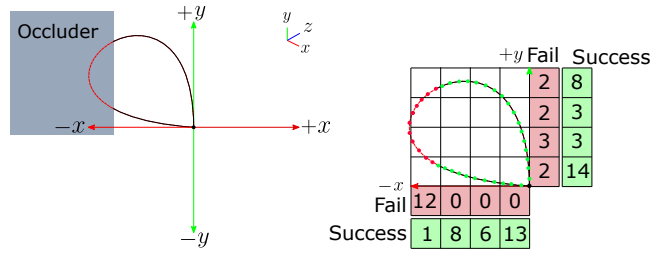


Figure 7: Projection of the success and fail of one Track on the four axis of resolution $R = 4$. a) Collision and occlusion detection b) Enumeration and projection of the success and fail samples on the axis

5.1 Camera animation quality

A proper camera animation to portray the motions of a target object should meet a set of nice properties, among which the most important are: avoid collisions with the scene, enforce visibility on the target object, while offering a smooth series of intermediate viewpoints to the viewer. To evaluate how much these properties are enforced along a camera animation $\mathbf{q}^i_j$, we propose to rely on a set of costs $C_k(t) \in [0,1]$:

Occlusions and Collisions To evaluate how much the target object is occluded from a camera position $q^i_j(t)$, we rely on ray-tracing. We firstly approximate the target object's geometry with a simple abstraction (*e.g.* a sphere). We secondly sample a set of points $s \in [0,N]$ onto this abstraction, which we position at the object's anticipated position $B^i(t)$. We thirdly launch a ray from the camera to each point s (see figure 4). We lastly note $R_s(t)$ the result of this ray-launch. In the mean time, we use the same ray to also evaluate if the camera is in collision (*i.e.* inside another object of the scene), by setting its value as:

$$R_s(t) = \begin{cases} 0 & \text{if Visible} \\ 1 & \text{if Occluded} \\ 2 & \text{if Collided} \end{cases}$$

We distinguish a collision from a simple occlusion as follow. By looking at the normal at the hit geometry, we know if the ray has hit a back face or a front face. When the ray hits a back face, $q^i_j(t)$ must be inside a geometry, hence we consider it as a camera collision. Conversely, when the ray hits a front face, $q^i_j(t)$ must be outside a geometry. If the ray does not reach s, we consider s as occluded, otherwise we consider it as visible.

Knowing $R_s(t)$, we define our collision and occlusion costs, as:

$$C_o(t) = \frac{1}{N}\sum_{s=0}^{N} \begin{cases} 1 & \text{if } R_s(t) = 1 \\ 0 & \text{Otherwise} \end{cases}$$

and

$$C_c(t) = \frac{1}{N}\sum_{s=0}^{N} \begin{cases} 1 & \text{if } R_s(t) = 2 \\ 0 & \text{Otherwise} \end{cases}$$

In our tests, we used $N = 20$.

Viewpoint variations Providing a smooth series of intermediate camera viewpoints requires to regulate changes on successive viewpoints. We hence propose to evaluate how much the viewpoint changes over time. We split this evaluation into two distinct costs:

one on the camera view angle, and one on the distance to the target object. Further, we define them for time steps δt.

Beforehand, let's introduce the view vector connecting the target object to the camera, on which we will rely. It is computed as:

$$D^i_j(t) = B^i(t) - q^i_j(t)$$

From this view vector, we define the view angle change as:

$$C_{\Delta_{\phi,\theta}}(t) = \frac{(D^i_j(t), D^i_j(t+\delta t))}{\pi}$$

In a way similar, we propose to rely on a squared distance variation, defined as:

$$\Delta d(t) = (\|D^i_j(t)\| - \|D^i_j(t+\delta t)\|)^2$$

We then define a cost on this distance change, which we further normalize as:

$$C_{\Delta d}(t) = 1 - E(\Delta d(t), \lambda)$$

where E is an exponential decay function, for which we set parameter λ to 10^{-4}.

Preferred range of distances One side effect of the above costs is that for large distances, changes on the view angle and distance will be less penalized. In turn, this will favor large camera animations. It is worth noting that, in the same way, placing the camera too close to the target object is also not desired in general. We should then penalize both behaviors. To do so, we propose to introduce a last cost, aimed at favoring camera animations where the camera remains within a prescribed distance range $[d_{min}, d_{max}]$. We formulate this costs as:

$$C_d(t) = \begin{cases} 1 & \text{if } \|D^i_j(t)\| \notin [d_{min}, d_{max}] \\ 0 & \text{otherwise} \end{cases}$$

5.2 Selecting a camera animation

Given the previously introduced costs, we now target to compute the cost of an entire animation, and select the most promising one.

In a first step, we define the total cost of a camera animation as a weighted sum of single-criteria costs integrated over time:

$$C = \sum_k w_k . \left[\int_{t_i}^{t_i+h} C_k(t)\ G(t-t_i, \sigma)\ dt \right]$$

where $w_k \in [0,1]$ is the weight of criterion k. G is a Gaussian decay function, where we set standard deviation σ to the value of $h/4$. We also slightly tune the decay to converge toward 0.25 (instead of 0). This way, we give a higher importance to the costs of the beginning of the animation, yet considering the end. Indeed, our assumption is that the camera will only play the first part of it (10% in our tests), while the remaining part still brings a longer term information on what could be a good camera path. We compute this total cost for any camera animation $q^i_j \in \mathbb{Q}^i$, and refer to it as C^i_j.

In a second step, we propose to choose the most promising camera animation for time horizon H^i, denoted as $\mathbf{q}^i$, as the one with minimum total cost, *i.e.* :

$$\mathbf{q}^i = \arg\min_j C^i_j$$

5.3 GPU-based evaluation

We have presented our evaluation metric on camera animations. However, some costs might be expensive to compute. In particular, computing occlusion and collision require to trace many rays (*i.e.* N rays, for many time steps, for hundreds of camera animations). We here focus more in detail on how we propose to perform computations in a very efficient way.

It is worth noting that many computations can be performed in parallel. The evaluation of camera animations are independent from each other. In a way similar, the evaluation of a cost at different time steps along a given animation are also independent. Hence, we propose to cast our evaluation of single costs into a massively-parallel computation on GPU.

Firstly, note that we only need to send the *animation space* (in orthonormal coordinate system) once to the GPU, when starting the system. Then, when we need to reposition the camera animation space, for horizon H^i, we simply update the 4x4 transform matrix M^i. And, from this data, one can straightforwardly compute any camera position $\mathbf{q}^i_j(t)$ for any time t.

Secondly, for occlusion and collision computations, we propose to rely on the recent RTX technology, allowing to perform real-time raytracing requests on GPU. We run $\frac{H^i}{\delta t}$ kernels per track, where each kernel launches N rays, one per sample s picked onto the target object. The result of these computations are stored into a 2D texture (as shown in figure 5), where the texture coordinates u and v map to one time step t and one animation of index j, respectively. Occlusion and collision costs are stored into two different channels.

Thirdly, we rely on a compute shader to compute all other costs, and combine them with occlusion and collision costs. This shader uses one kernel per camera animation. It stores the total cost of all animations into a GPU buffer, finally sent back to CPU where we perform the selection step.

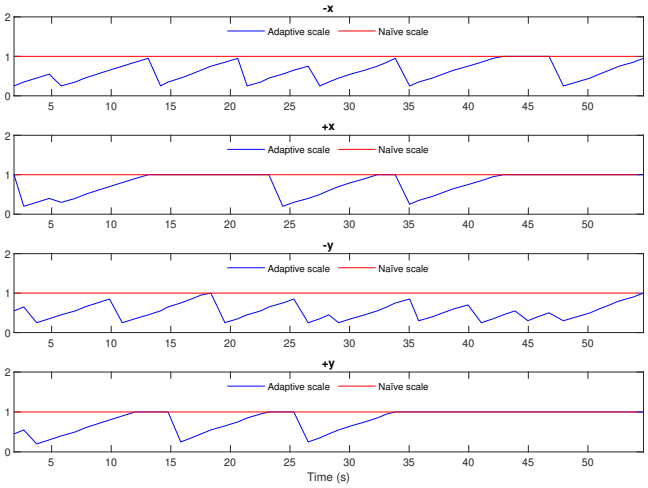
6 Dynamic Trajectory Adaptation

Until now, we have considered a simplified configuration, where we evaluate the *animation space* and select one camera animation for one given time horizon H^i. We now need to consider two other requirements. First, the camera should be animated to track the target object for an unknown duration, larger than h. Changes in the target behavior may also occur, due to interactive user inputs. Second, for any horizon H^i, some camera animations could be in collision with the scene, or the target could be occluded. This would prevent finding a proper animation to apply. In other words, the space of potential camera animations should be influenced by the surrounding scene geometry. Hereafter, we explain how we account for these requirements.

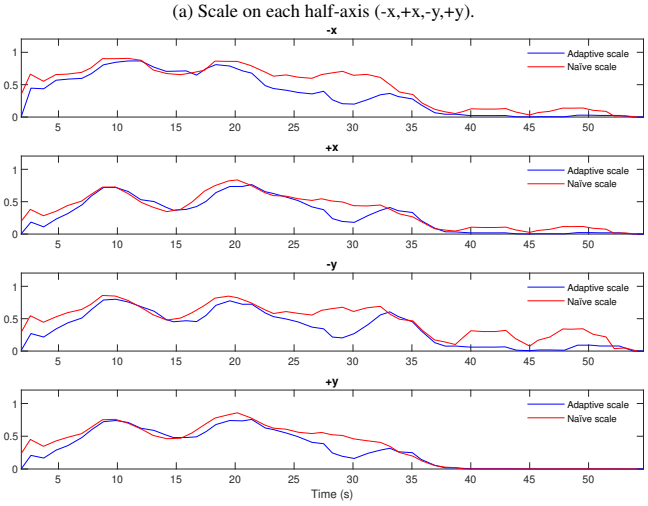
6.1 User inputs and interactive update

We here assume the camera is currently animated along curve $\mathbf{q}^i$. We then need to compute a new animation, for a new time horizon H^{i+1} in two cases. First, when the target's behavior has changed, this makes the currently played animation not valid for future time steps. Second, when the camera animation as reached a specific duration. In a way similar to motion-predictive control, we indeed consider the target behavior, as well as the collision and occlusion computations, less reliable after a certain anticipation duration. This in particular allows to handle dynamic collisions and occlusions. This update is specified by the user as a ratio of progress along animation $\mathbf{q}^i$. In our tests the horizon duration is $h = 5$ seconds and the update ratio is 0.1. In turn, the new horizon generally starts at $t_{i+1} = t_i + 0.1h$, while we set $\mathbf{q}^{i+1}_{start} = \mathbf{q}^i(t_{i+1})$.

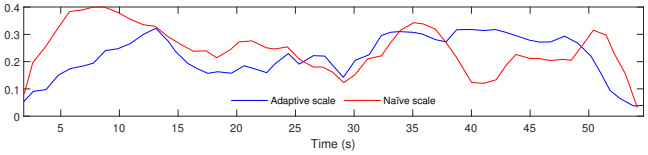
Knowing that an update is required, we iterate on the overall process explained earlier, for next horizon H^{i+1}. We select a new goal viewpoint (*i.e.* $\mathbf{q}^{i+1}_{goal}$), and update the transform matrix (*i.e.* M^{i+1}) to position the camera animation space $\mathbb{Q}^{i+1}$. We then need to evaluate all camera animations in $\mathbb{Q}^{i+1}$. To do so, we compute all costs presented in section 5. However, this is not enough, as we also need to enforce continuity between animation q^i and the animation $\mathbf{q}^{i+1}$ that



(a) Scale on each half-axis (-x,+x,-y,+y).



(b) Percentage of fails on each half-axis (-x,+x,-y,+y).



(c) Best animation score.

Figure 8: Comparison of our system with an adaptive scale, or with a naïve scale, applied on the camera animation space.

is to be selected. To do so, we rely on an additional criteria designed to favor a smooth transition between consecutive animations:

Animation transitions this cost penalizes abrupt changes when transitioning between two camera animation curves. Our idea is to penalize a wide angle between the tangent vector to camera animation $\mathbf{q}^i$ and the tangent vector to animation $\mathbf{q}_j^{i+1} \in \mathbb{Q}^{i+1}$, at connection time t_{i+1}. We write this cost as:

$$C_{i,i+1}(j) = \frac{(\dot{\mathbf{q}}^i(t_{i+1}), \dot{\mathbf{q}}_j^{i+1}(t_{i+1}))}{\pi}$$

We then rewrite the selection of camera animation q^{i+1} as:

$$\mathbf{q}^{i+1} = \arg\min_j \left[C_j^{i+1} + w_{i,i+1}\, C_{i,i+1}(j) \right]$$

where $w_{i,i+1}$ is the relative weight of the transition cost with regards to other costs.

6.2 Adapt to scene geometry

We would also like to adapt our camera animation space to the scene geometry. To this aim, on the one hand we seek to offer a camera animation space which exhibits as few collisions and occlusions as possible. On the other hand, we also seek a space which is not too restricted, *i.e.* covering as much as possible the free space between the target behavior and the scene geometry.

To do so, while we evaluate the quality of camera animations for an horizon H^i, we also take the opportunity to analyse how much collisions and occlusions occur. In turn, this will allow to know if the free space is well covered (or not enough). We then propose to dynamically rescale the camera animation space to make it grow or shrink in the next time horizon H^{i+1}. This rescaling applies when we update the transform matrix M^{i+1}, and on the x and y axes only. It is worth noting that the free space might not be symmetrical around the target behavior (as illustrated figure 6). Indeed, this free space might for instance be larger (or smaller) on the left than on the right of the target. The same applies to the free space above or below the target. Consequently our idea is to compute four scale values, on all four directions $\{-x, +x, -y, +y\}$. For any camera position along a camera animation, we then apply either two of them, depending on the sign of the position's x and y coordinates in the non-transformed animation space.

We propose to process in the following way. We firstly leverage the occlusions and collisions evaluation to store additional information: we count fails and successes along each axis. We consider a launched ray along a camera animation (*i.e.* from the camera position at a given time step) as a fail if it is marked as occluded or collided, and as a success if not. We secondly store this information in height arrays: for each half-axis (*e.g.* $+x$ or $-x$), we count successes in one array, and fails in another array. We further discretize this half-axis by using a certain resolution R, to output two histograms, of fails and successes (as illustrated in figure 7). Note that R here define the scale precision on each axis. We lastly use both histograms to compute the new scale to apply. We compute the indices i_f and i_s of the medians of both arrays (fails and successes, respectively). By comparing them, we define how much we should rescale animations along this half-axis. If $i_s < i_f$, we consider that there are too many fails, and multiply the current scale by i_f/R, to shrink animations. Otherwise, we consider the free space is not covered enough, and apply a passive inflation to the current scale. The aim of this inflation is to help return to a maximum scale value, when the surrounding geometry allows for large camera animations.

7 Implementation and Results

7.1 Implementation

We implemented our camera system within the Unity3D 2019 game engine. We compute our visibility and occlusion textures through raytracing shaders provided with Unity's integrated pipeline, and perform our scores for all sampled animations and timesteps through Unity Compute Shaders. All our results (detailed in section 7.2) have been processed on a laptop computer with a Intel Core i9-9880H CPU @ 2.30GHz and a NVIDIA Quadro RTX 4000.

7.2 Results

We split our evaluation into three parts. We firstly validate our adaptive scale mechanism. We secondly evaluate the robustness of our system, by comparing its performances when using a different number or set of reference camera animations. We thirdly validate the ability of our system, mixing local and global planning approaches, to outperform a purely local camera planning system. To do so, we compare results obtained with our system, and the one of Burg *et al.* [1], on the same test scenes.

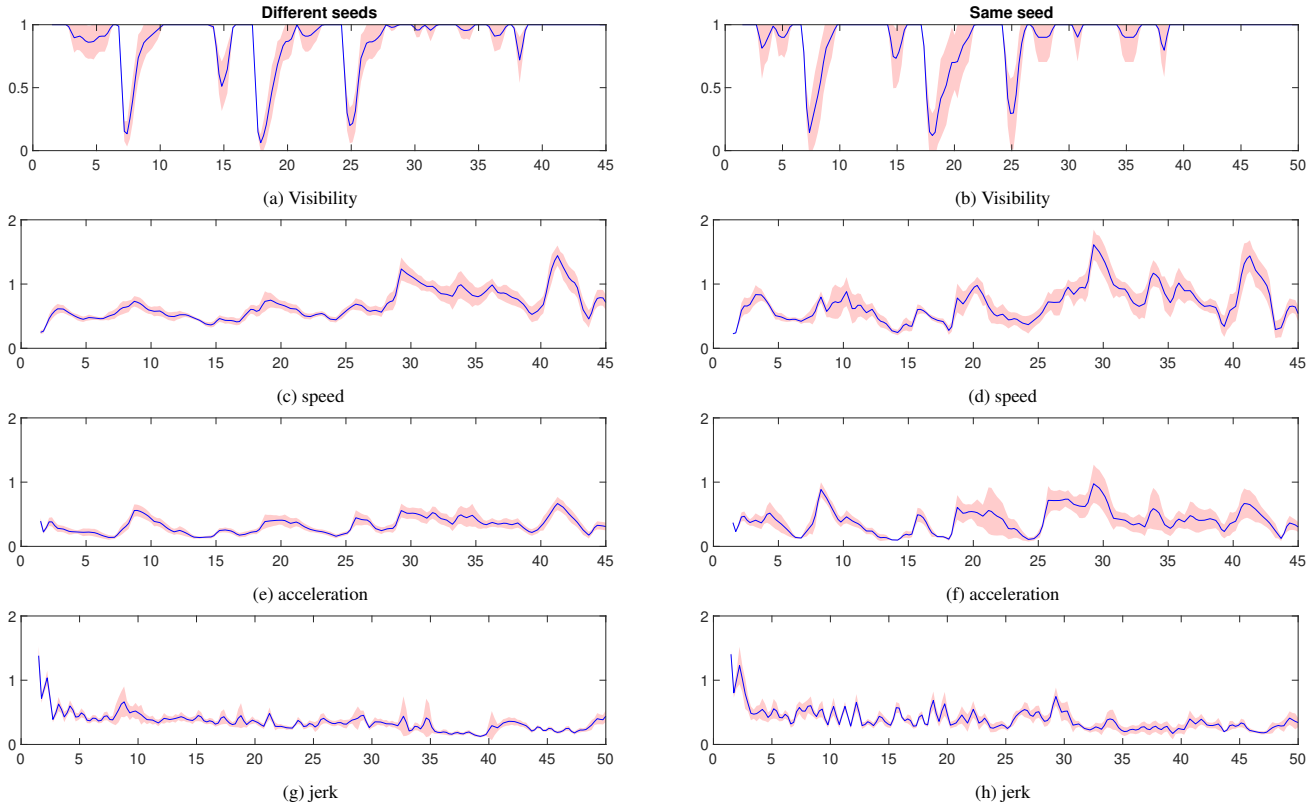


Figure 9: Results for multiple runs, each using a randomly generated camera animation space. This space is sampled with uniform distribution, with 2400 sample camera animations (Hermite curves). Each plot shows the mean value over time (blue), with a 95% confidence interval (red). Left: results for 22 runs using different seeds. Right: results for 10 runs using the same seed.

To validate our adaptive scale, we study its impact on the quality of the animation space. For the other evaluations, we compare camera systems with regards to two main criteria: how much the camera maintains visibility on the target object, and how smooth camera motions are. We compute visibility by launching rays onto the target object, and calculating the ratio of rays reaching the target. A ratio of 1 (respectively 0) means that the target is fully visible (respectively fully occluded). When relevant, we additionally provide statistics on the duration of partial occlusions. We then compare the quality of camera motions through their time derivatives (speed, acceleration and jerk), which provide a good indication of motion smoothness.

Our comparisons have been performed within 4 different scenes (illustrated in the accompanying video). We validated our system by using (i) a **Toy example scene**, where the target is travelling a maze, containing several tight corridors with sharp turns, an open area inside a building, and a ramp. We then performed the comparisons with the technique of Burg *et al.* [1], by using two static scenes and a dynamic scene, which the target goes through: (ii) a scene with a set of columns and a gate (**Columns+Gate**), (iii) a scene with set of small and large spheres (**Spheres**) and (iv) a fully dynamic scene with a set of randomly falling and rolling boxes, and a randomly sliding wall (**Dynamic**). To provide fair comparisons, in the dynamic scene, we pre-process the random motions of boxes and of the wall. As well, for all tests in a scene, we play a pre-recorded trajectory of the target avatar, but let the camera system run as if the avatar was interactively controlled by a user.

7.2.1 Impact of adaptive scale

We validate our adaptive scale (section 6.2) by comparing results obtained (i) when we compute and apply the *adaptive scale* on all 4 half-axes ($-x$,$+x$,$-y$,$+y$), and (ii) when we simply apply the same scale as for the z axis (which we will call the *naive scale* technique). We processed our tests by using the toy example scene. For each technique, each time we evaluate a new set of camera animations, we output the new scale values and the ratio of fails on each half-axis. As well, we output and plot the mean cost of the 5 best animations in this set. Results are presented in figure 8.

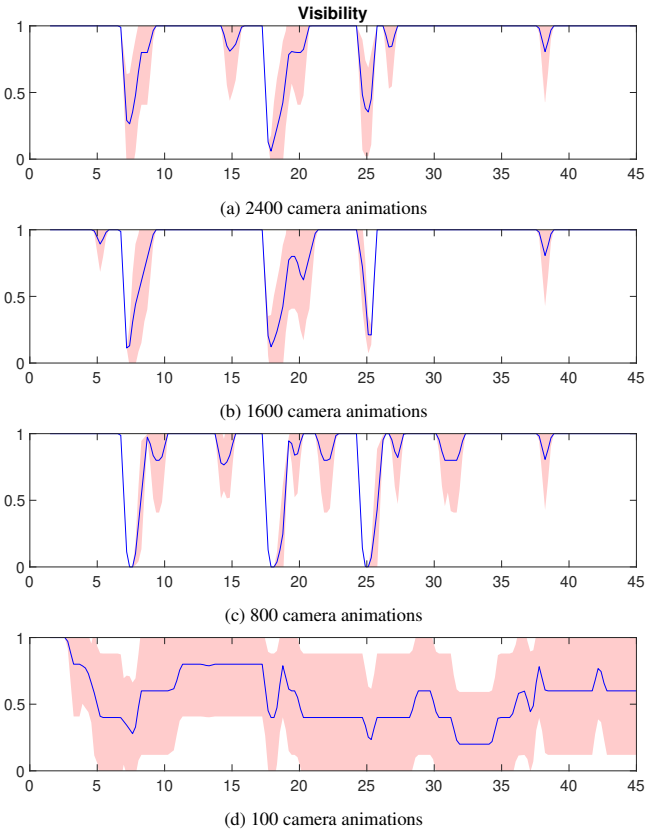
Figure 8a shows how much our mechanism tightens the animation space (compared to the naive scaling technique) when the avatar is entering corridors, and grows back to the same scale when the avatar reaches less cluttered areas (*e.g.* in the open interior room, or the outdoor area). As expected, our mechanism allows to adapt the scale on half-spaces in a non-symmetrical way. As shown by figure 8b, with our adaptive mechanism, the scaled animation space also exhibit less fails than using the naive scale technique. As well, as shown by figure 8c, it allows finding animations with lower cost, most of the time. One exception is between 40s and 50s, where the camera configuration isn't the same because the scale is different. In the naive case the camera is high above the character while in the adaptive case, the camera is closer to the ground, thus the scores is not relevant in this case because the two configurations are way to different to be compared.

In the next evaluations, we consider that the adaptive scale mechanism is always activated.

7.2.2 Robustness

We also study the robustness of our system regarding our randomly generated camera animation space.

In a first step, we evaluate how performances vary if we run our real-time system multiple times, on the toy example scene. We also



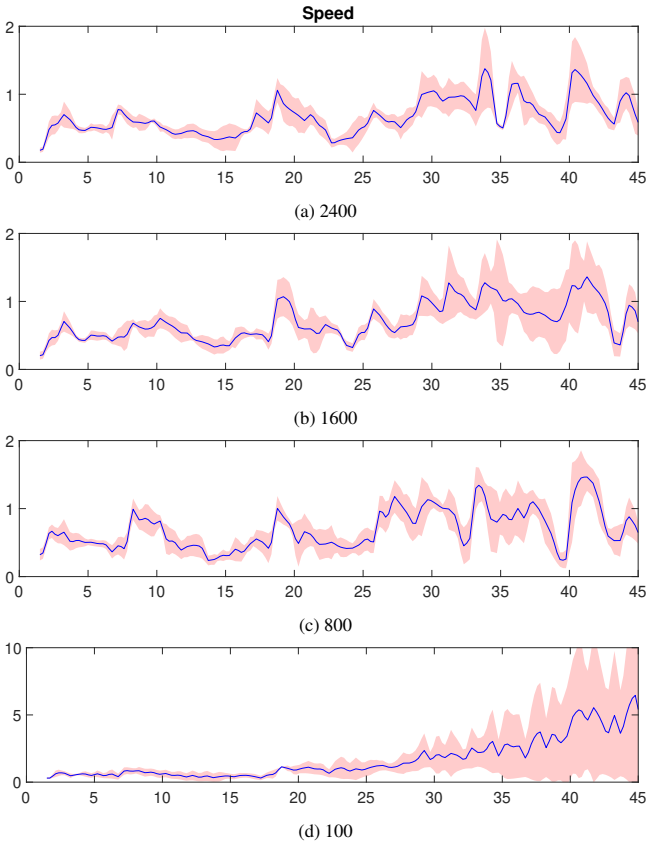
(a) 2400 camera animations

(b) 1600 camera animations

(c) 800 camera animations

(d) 100 camera animations

Figure 10: Visibility when varying the number of sampled curves in our camera animation space.



(a) 2400

(b) 1600

(c) 800

(d) 100

Figure 11: Camera speed when varying the number of sampled curves in our camera animation space.

consider two cases: (i) using the same seed for every run (*i.e.* the same animation space is used), and (ii) using a new seed for every run (*i.e.* a new animation space is randomly sampled for each run). For each case, we sample a set of 2400 animations. Results are presented in figure 9. As it shows, with as many sampled animations, all runs lead to very similar results, both on the visibility enforcement, and on the camera motion smoothness. Differences are mainly due to variations in the actual framerate of the game engine, hence the rate at which the system takes new decisions.

In a second step, we evaluate how the size of the animation space (*i.e.* the number of sampled animations) impacts performances. We ran our system with 4 different sizes: 2400, 1600, 800 or 100 animations. For each size, we performed 5 runs with random seed, and combined the results in figures 10, 11 and 12. It shows that lowering the size (at least until 800 animations) still allows good performances. Our camera system is able to find a series of camera animations maintaining enough visibility on the target object, through smooth camera motions. As we expected, for 100 animations, our system's performances are poor: it becomes harder to find animations with sufficient visibility and ensuring smooth camera motions. Our intuition behind this result is that as soon as the size become too small, the distribution of tangents becomes very sparse, hence breaking our assumption of a uniform sampling. If the *animation space* does not sufficiently cover the free space, this prevents the exploration of a wide range of possible camera animations.

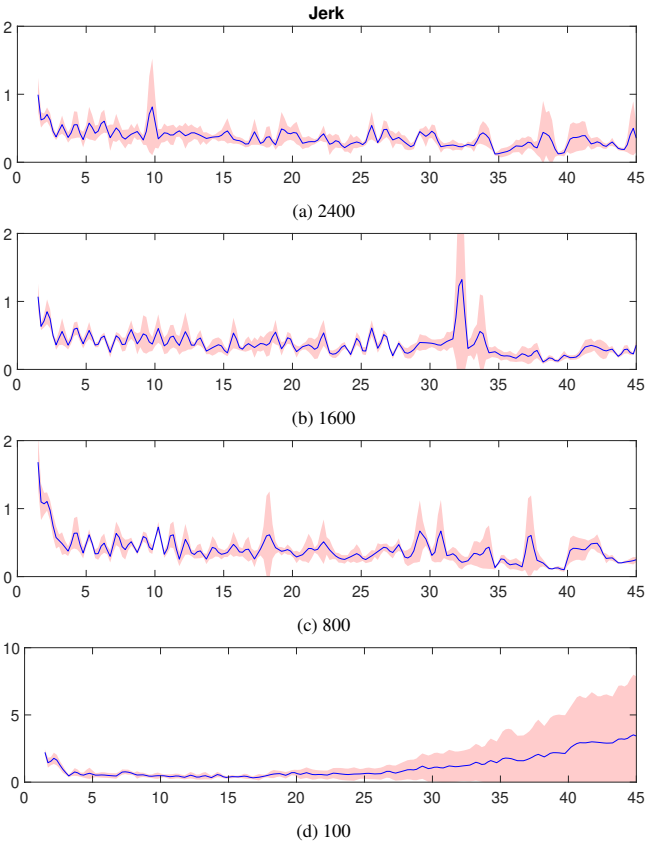
7.2.3 Comparison to Burg 2020

We also compare our system, mixing local and global planning approaches, to a purely local camera planning system. To this aim, we have run our proposed camera system, and the local camera planning system of Burg *et al.* [1], in 3 different scenes: two static scenes (**Columns+Gate** and **Spheres**) and a fully dynamic scene (**Dynamic**). The **Columns+Gate** is the same as in [1], where the avatar is moving between some columns and go through a doorway. In the **Spheres** scene, the avatar is travelling a scene filled with a large set of spheres, which makes it moderately challenging for the camera systems. In the **Dynamic** scene, the avatar must go through a flat area, where a set of boxes are randomly flying, falling, rolling all over the place, and a wall is randomly sliding. This makes it challenging for camera systems to anticipate the scene dynamics and find occlusion-free and collision-free camera paths.

In our camera system, we used 2400 animations the recomputation rate is set to 0.25s and the adaptive scaling is on. We present results of our tests in figures 13, 14, 15, 16, and 17.

We firstly compare camera systems along their ability to enforce visibility on the target object (figure 13). Our tests show that for moderately challenging scenes, both lead to relatively good results. Few occlusions occur. However, for a more challenging scene (**Dynamic**), our system outperforms Burg *et al.* 's system. Even if occlusions may occur more often, the degree of occlusion is lower (figure 13b). Moreover, for all 3 scenes, when partial occlusions occur, they are shorter when using our system (figure 13c). This is explained by the fact that when no local solution exist, our system can still find a locally occluded path respecting other constraints, and leading to a less occluded area. This demonstrates our system's ability to better anticipate occlusions, and especially in dynamic scenes.

We secondly compare the smoothness of camera motions in both camera systems. Figure 14 presents, side-by-side, the distributions of speed, acceleration and jerk when using each system. We also



(a) 2400

(b) 1600

(c) 800

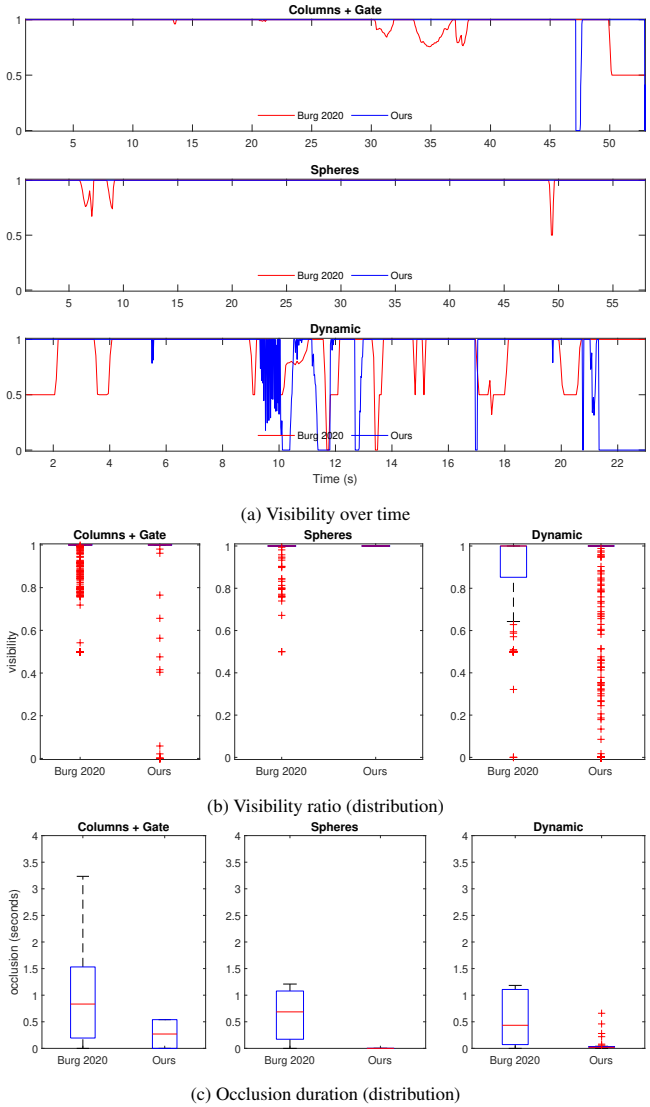
(d) 100

Figure 12: Camera jerk when varying the number of sampled curves in our camera animation space.

provide the speed, acceleration and jerk along time, in figures 15, 16, and 17. One observation we make is that Burg *et al.* 's system leads to lower camera speeds, as it restricts itself to simply following the avatar. In our camera system, the camera is allowed to move faster, to bypass the avatar when visibility or another constraint may be poorly satisfied. Yet, our system provides smoother motions (*i.e.* less jerk). One explanation is that local systems often need to steer the camera from local minima (*e.g.* low visibility areas). A side effect is that it may lead, for successive iterations, to an indecision on which direction the camera should take to reach better visibility. In turn, this leads to frequent changes in camera acceleration (hence higher jerk). Conversely, our system has a more global knowledge on the scene, allowing to more easily find a better path, which avoids sacrificing the smoothness of camera motions.

8 Discussion and Conclusion

Our system presents a number of limitations. Despite the ability to evaluate thousands of trajectories, strongly cluttered environments remain challenging. As smoothness is enforced, visibility may be lost in specific cases, and designing a techniques that could properly balance between the properties to handle specific cases need to be addressed. Also while the dynamic scale adaptation does improve results by compressing the trajectories in different half spaces, low values in scales prevent the camera from larger motions where necessary. A future work could consist in biasing the sampling in the *animation space* in order to adapt the space to typical local topologies of the 3D environment. Despite the limitations, the proposed work improves over existing contributions by proving an efficient camera tracking technique adapted to dynamic 3D environments and does not require heavy roadmap precomputations.



(a) Visibility over time

(b) Visibility ratio (distribution)

(c) Occlusion duration (distribution)

Figure 13: Comparison between our system and Burg *et al.* [1], regarding the target object's visibility (a)(b) and, when not fully visible, the duration of partial occlusion (c).

References

[1] L. Burg, C. Lino, and M. Christie. Real-time anticipation of occlusions for automated camera control in toric space. In *Computer Graphics Forum*, volume 39, pages 523–533. Wiley Online Library, 2020.

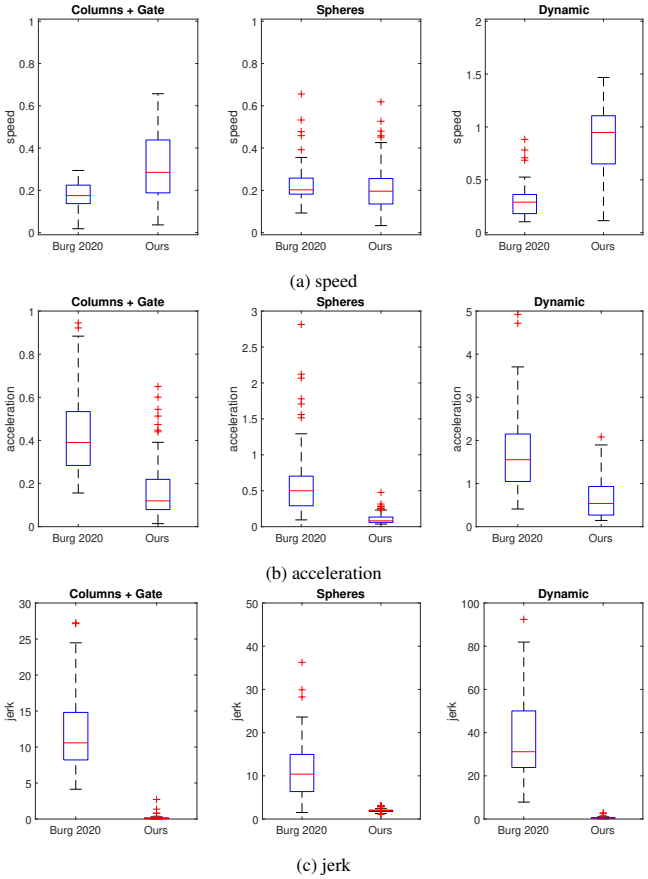
[2] M. Christie, J.-M. Normand, and P. Olivier. Occlusion-free camera control for multiple targets. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2012.

[3] M. Christie, P. Olivier, and J.-M. Normand. Camera control in computer graphics. In *Computer Graphics Forum*, volume 27, pages 2197–2218. Wiley Online Library, 2008.

[4] N. Halper, R. Helbing, and T. Strothotte. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. In *Computer Graphics Forum*, volume 20, pages 174–183. Wiley Online Library, 2001.

[5] A. Jovane, A. Louarn, and M. Christie. Topology-aware camera control for real-time applications. In *Motion, Interaction and Games*, pages 1–10. 2020.

[6] C. Lino and M. Christie. Intuitive and efficient camera control with the toric space. *ACM Transactions on Graphics (TOG)*, 34(4):1–12, 2015.



(a) speed

(b) acceleration

(c) jerk

Figure 14: Comparison between our system and Burg *et al.* [1], regarding the camera speed (a), acceleration (b) and jerk (c) distributions.

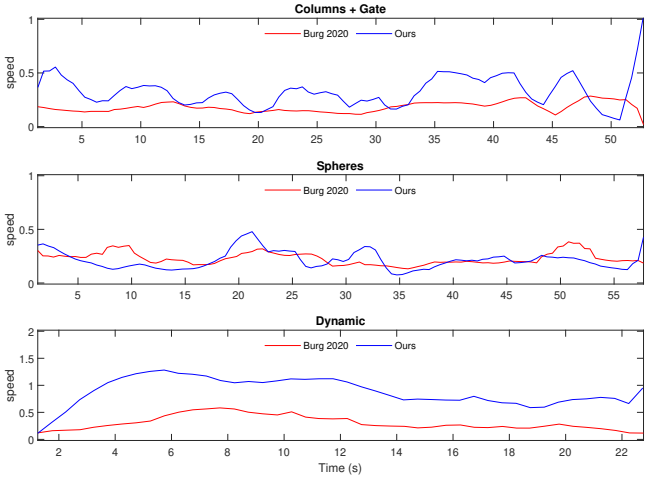


Figure 15: Speed along time, for our camera system (blue) and the one of Burg *et al.* [1] (red)

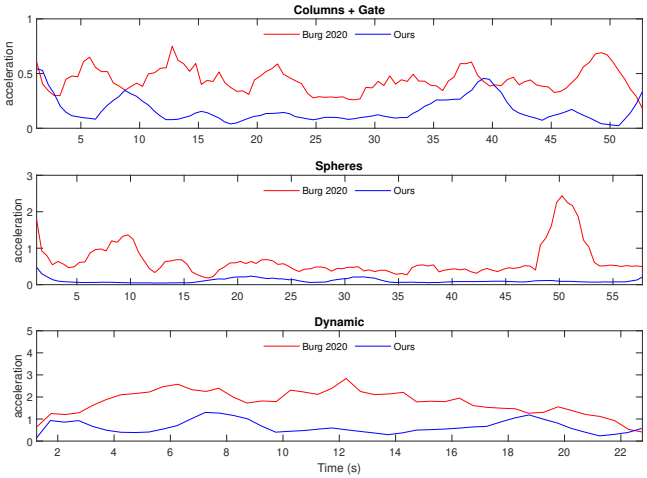


Figure 16: Acceleration along time, for our camera system (blue) and the one of Burg *et al.* [1] (red)

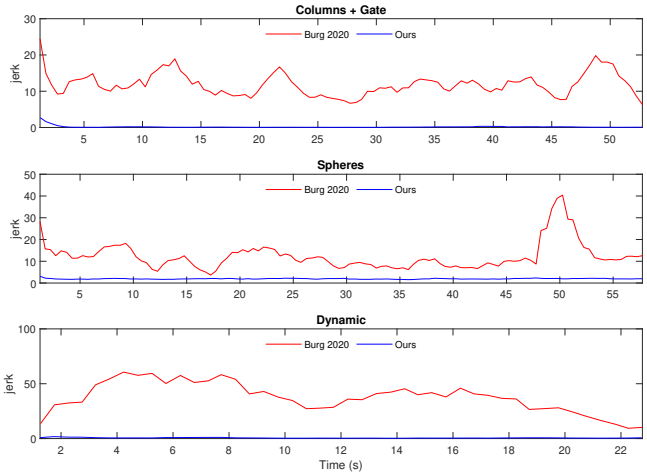


Figure 17: Jerk along time, for our camera system (blue) and the one of Burg *et al.* [1] (red)

[7] C. Lino, M. Christie, F. Lamarche, G. Schofield, and P. Olivier. A real-time cinematography system for interactive 3d environments. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 139–148. Eurographics Association, 2010.

[8] A. Litteneker and D. Terzopoulos. Virtual cinematography using optimization and temporal smoothing. In *Proceedings of the Tenth International Conference on Motion in Games*, pages 1–6, 2017.

[9] T. Nägeli, L. Meier, A. Domahidi, J. Alonso-Mora, and O. Hilliges. Real-time planning for automated multi-view drone cinematography. *ACM Transactions on Graphics (TOG)*, 36(4):1–10, 2017.

[10] D. Nieuwenhuisen and M. H. Overmars. Motion planning for camera movements. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 4, pages 3870–3876. IEEE, 2004.

[11] T. Oskam, R. W. Sumner, N. Thuerey, and M. Gross. Visibility transition planning for dynamic camera control. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 55–65, 2009.

[12] R. Ranon and T. Urli. Improving the efficiency of viewpoint composition. *IEEE Transactions on Visualization and Computer Graphics*, 20(5):795–807, 2014.

[13] P. O. Scokaert and D. Q. Mayne. Min-max feedback model predictive control for constrained linear systems. *IEEE Transactions on Automatic control*, 43(8):1136–1142, 1998.