

SpaceCam: a tool for machinima cinematography and editing

Thomas James Lodato
thomas.lodato@gmail.com

ABSTRACT

SpaceCam is a cinematography and editing system for machinima production. Relying on a methods of comprehensive logging of data, SpaceCam provides a manipulable process of placing cameras and visualizing action in a demo file. The rethought metaphor for editing extends beyond perspectival constraints to begin the discussion of how to better conceive of digital cinematography and editing.

Keywords

SpaceCam, cinematography, editing, demo file, data log

1. INTRODUCTION

Machinima is currently more limited by a conceptual framework than it is by technical restrictions. A prime example is the name machinima itself. In practice, it consists of moving images rendered in real-time animation through a videogame engine. Hence, the name resembles the typically definition: a genre falling between theater, cinema, and (digital) animation. Machinima, however, is far from simply a sub-genre or mixed-media practice. And, while machinima resembles (and even uses) these media, the amalgamated definition implants a subverting thought in practitioners and theorists heads; that is, machinima is bound by its derivative fields.

As a result, many of the tools for machinima have been remediations of legacy production models and methodologies. The way things have been done in theater, and in animation, and in film have formed the cogent basis for machinima. While the translated acting, directing, editing, and a myriad of other tasks, enact the affordances and conceptual frameworks with which users are comfortable, they also impose the severe limitations of real-world concerns.

SpaceCam introduces a re-conceptualization of the process of editing. This model stems from the ability to log data within a real-time engine rather than record perspectives. Three-dimensional action is transformed from embedded viewpoints of past action to dynamic performances of current action. SpaceCam strips away many of the legacy conventions of nonlinear editing software and replaces them with affordances particular to real-time engines, hence positing a specific editing system for 3D virtual environments.

2. THEORY

2.1 Core Questions

SpaceCam stems from two fundamental questions:

(1) What built-in features of real-time engines can be appropriated for capturing scene data?

(2) How can this data be differently visualized to take into account the malleability of digital environments?

These questions led to the demo file and data log.

2.2 The Demo and The Log

A demo is a feature of a real-time engine that has been part of machinima from the onset of production. First appearing in 1994 with the release of *DOOM*, Paul Marino explains,

[T]his feature allowed a gamer to record the action of the game as it occurred, [which] could be played back in real-time through the game engine. [I]t was ... incredibly efficient because it captured only the positions, orientations, and movements of the game instead of full frames of information. [1]

Marino highlights a few important technical, and so conceptual, differences between the demo and alternate methods of capturing (videogame) action. Demo recording, in its efficiency, does not (necessarily) capture the world that surrounds a player's actions. Instead, the demo can create a comprehensive, yet *mise-en-scene*-less, record of the location, rotation, and movement of an avatar; this is known as the data log. The log is essentially pure action, while the demo contextualizes that action on playback. The landscape, lighting, even the avatar's appearance, are not stored directly. The demo on playback recalls the static elements of the world to visualize the log.

In this exists one of the loopholes of the log file. As the motion is the only captured element, there is potential to change the context of that action. A blank soundstage could be replaced or manipulated after acting. While this was not the focus of SpaceCam, the methodology behind logging (as opposed to recording) affords future work here.

Upon visualization, the log is channeled back into the game engine and re-acted automatically. The combination of logging and replaying is what is referred to as the demo. Notably, no where in this process does camera position enter the log or the demo. In the re-acted scene, users (typically) can float in third-person through the world or jump into any viable first-person perspective.

The action in the demo is not arrested from the world. This is the second distinct feature of the demo and log. In capturing a gameplay sequence, the log also takes in data of the animations (i.e. actions) that take place. When the user runs a demo--that is, visualizes the log--the game enacts the sequenced actions of an avatar. Technically, the engine renders actions through built-in physics. That is, a jump in a demo not equivalent, but identical, to

a jump during a game. The sole difference resides in from where the initiation of that action stems--during gameplay, from an interactor; during demo playback, from a data file. The demo playback is ostensibly a living environment, albeit following a single evolutionary trajectory, namely the log file.

2.3 How is this different?

The recorded action and environment in cinema and the pre-rendered situations of digital animation starkly oppose the demo-log methodology in a few distinct ways. While the moving images in all cases are similar, these modes of recording or rendering sensibly diverge in their relationship with action and liveness.

First, both cinema and animation arrest action from space and time. A single moment in a film and an animation is a frame. A single moment in a log, on the other hand, is not able to be separated from the previous and subsequent moment the way a frame can be. A moment in a log is a continuation of previous moments rather than a sequence of contiguous-and-related frames.

In this way, the demo playback can be equated to a (theatrical) performance. Philip Auslander in discussing *Listening Post* brings up what constitutes a human, and so a machine, performance.

[T]he fundamental difference between human performers and machine performers is that whereas the former have the potential to exercise both technical and interpretive skills the latter have only technical skills at their disposal. [2]

Auslander's view places the sequential, albeit automatic, retrieval of information in the plane of technical-skill performance, comparable to a person playing a piece of music.

The demo, unlike visuals recorded through film or animation, does not simply retrieve information at the display level, such as projecting a film. Instead, the demo retrieves data at the command and processing level, regenerating the imagery on each playback. Just as a live theatrical performance uses the same blocking, setting, sequencing, but must be acted each night, the demo must be analogously acted by the machine. In the legacy media, repeating viewings are not performative as much as they are informative.

3. PREVIOUS WORK

3.1 CamBot

CamBot implements "a lightweight artificial intelligence system" that, based on a user-generated script, "blocks characters, identifies possible shot compositions, and edits the available shots into a final *reel*." [3] (original emphasis) Furthermore, CamBot generates a visual component to the story (i.e. a machinima piece) that abides by established conventions of film and television. The application allows the user to script the dialogue and action (and so controlling mood, tempo, narrative, etc.) and assumes control at the shot and editing level. To parallel traditional roles in cinema, the user is the script writer and the director and CamBot is the cinematographer and editor.

In critique of CamBot, its prescriptiveness and finiteness are serious weaknesses in terms of its expressivity. No matter the script content, weighted values determine the shot/editing choices from a limited set of pre-defined shots and cuts. It should be noted, though, that CamBot was not created to make machinima *per se*, but to explore how AI could be implemented in machinima's creation--and visual media on the whole--as an automating tool. This end contributes to the underlying narrow expressive functionality, limited by the difficulties of AI story generation. Users are pigeonholed (to a greater or lesser extent based on map layout, script content, etc.) into prescriptive Hollywood conventions that are utilized due to their familiarity with visual tradition.

CamBot, in terms of implementation, raises an important point about modularity. The application is not tied to a particular real-time 3D engine (though it currently uses Unreal Tournament 2003). This is important for two reasons for both CamBot and SpaceCam. First, when a tool is tied to a particular engine (such as RypelCam to the Unreal Engine), the latest version of that game does not require a newer version of the tool. It would be comparable to a large hammer only being able to hit big nails, but unable to tap small ones.

Secondly, as the application intends to explore visual stories, it should not be linked to a specific *type* of visual story. In a parallel sense, moving pictures can be made with hand-held cameras and high-definition cameras, analog and digital formats, black-and-white and technicolor film stocks, while still remaining within the medium. Hence, a tool limited by technologic constraints is equally limited in its expressive constraints. In this sense, CamBot flourishes.

3.2 ViGLS

ViGLS "automatically produces real-time visualizations of the summarized actions which are extracted based on cognitive models of summarization." [4] ViGLS is designed not to make machinima; it consolidates action into a relevant summary plot. In logging the entirety of action, the program algorithmically weights actions to produce a more concise log that will be represented in a summarizing movie. Furthermore, ViGLS visualizes the relevant narrative through occlusion-free camera selection (similar to CamBot).

Though ViGLS exploits logging, the narrow intent--summarization--limits its potential application to expressive forms of machinima. Equally limiting (as with CamBot) is the algorithmic generation of the final movie. The role of editor and cinematographer are played by the computer, while the actors are scripted and directed by the user.

3.3 RypelCam-

RypelCam is a camera-control mod created for the Unreal Engine. The tool allows users to flexibly manipulate the camera through a logged scene, as well as make minor directorial decisions, such as

hiding weapons and switching between first- and third-person viewpoints.

The first, and most serious, limitation of RypelCam are its dependency on UnrealScript. As can be seen from release notes, it must be updated as frequently as the Unreal Engine is updated. While this is every few years, the dependency on UnrealScript locks the users into using Unreal Tournament for their machinima production. All machinima is then funneled out of a single game, which categorically disregards the potential expressivity of the art form.

Secondarily, RypelCam does not provide much more functionality, in terms of camera control, than the built-in demo logging. The user controls the camera in a standard first-person perspective. Hence, the editor cannot be directly aware of action in an alternative location within a map. If RypelCam were to be used with a demo, then a log of actions would have to be generated and employed before adequate shooting could occur; if used in real-time, live correspondence with all actors. The gracelessness of such a production method both seriously impairs the production process and deters potential users, and provides little added functionality.

4. SpaceCam

SpaceCam constitutes a three-part process:

- (1) Logging
- (2) Visualizing
- (3) Placing cameras

4.1 Logging

The logging process for SpaceCam relies on two parts: the game type (in UnrealScript) and the SpaceCam (in Java).

Initially, the idea was to use a lightweight mutator of Unreal Tournament 2004 to capture the requisite information (position, rotation, etc.). Due to technical constraints, a game type is used to gather data. This poses a problem in two ways. First (and most important), it is required on all computers that are being used as actors in a scene. This reduces the speed and the flexibility of implementation. Secondly, it heavily depends on scripting in Unreal Tournament 2004. This means upgrades will not simply be on Java side as originally hoped.

The actual process consists of simply finding locations. All pawns in a scene are tracked during the course of the action by running the SpaceCam game type. Simultaneously, the SpaceCam program in Java collects information being sent out of Unreal. The user records the information to a text file when the scene is over.

One of the biggest difficulties in the creation of SpaceCam came from this stream of information. Since data is sent as quickly as possible (in order to have the finest granularity of motion), the data stream quickly became tied up. This results in either sparse

data on the whole or a dominant data stream (occluding another almost entirely). SpaceCam intends to be used in multiplayer environments, this seriously hindered fuller implementation and development in these virtual spaces.

4.2 Visualizing

After the positions are logged, the exterior program visualizes the paths of the pawns in space. The user can track position through three dimensions (the x-y plane directly; z-coordinates through color coding), as well as through time.

The visual field of the paths is the only displayed space. That is, the SpaceCam panel is bounded by where an actor actually goes. In retrospect, this may not have been the best choice of visualizing paths, though it is the simplest. In the subsequent step of placing cameras, the positions sent to Unreal shift if the bounds change. This is obviously not an issue for demo playback, but it is for real-time uses of SpaceCam. On the positive side, the conceptual implications--space is understood (and important) only through performance--provide an argument for better understanding how virtual spaces are performance spaces.

4.3 Placing Cameras

Once the paths are visualized, cameras can be placed on the Java panel and sent to Unreal when triggered. The path defined for each camera can be changed by dragging on the Java side.

The difficulty of the project is the demo file. The game type does not allow a demo playback to occur with manipulation of cameras. This is a serious blow to the implementation, though insubstantial to the theoretical backing. Given an engine that has more open demo files, full camera manipulation and recording could potentially be incorporated.

5. CONCLUSION

SpaceCam's ambitious attempt to rethink machinima production has left a great deal of room for future extensions. Save a full implementation of the method described for SpaceCam, machinima would be greatly aided by tools that spatialize and re-render other parts of the machinima process such as directing, blocking, lighting, and sound.

6. REFERENCES

- [1] Marino, Paul. "Chapter 1: Understanding Machinima" in *3D Game-Based Filmmaking: The Art of Machinima*
- [2] Auslander, Philip. "At the *Listening Post*, or, do machines perform?" in *International Journal of Performance Arts and Digital Media, Volume 1 Number 1*. (2005)
- [3] Elson, David K. and Riedl, Mark O. "A Lightweight Intelligent Virtual Cinematography for Machinima Production" from *Association for the Advancement of Artificial Intelligence* (aaai.org) 2007 [http://www.aaai.org/Papers/AIIDE/2007/AIIDE07-002.pdf]
- [4] Cheong, Yun-Gyung, et al. "Automatically Generating Summary Visualizations from Game Logs" from *Association for the Advancement of Artificial Intelligence* (aaai.org) 2008 [http://www4.ncsu.edu/~bbae/publication/AIIDE3CheongY65.pdf]

[5] RypelCam [http://rypelcam.net/_3vo/]