Zuzen, a Cloud-based Framework for Automated Machinima Generation

Samuel Munilla
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
srmunill@ncsu.edu

R. Michael Young
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
young@csc.ncsu.edu

ABSTRACT
The Zuzen framework is an intelligent tool set for assisting in the generation of machinima. With Zuzen, users that are novice cinematographers do not need to use complex movie-making tools. Rather, they only need to specify a set of high-level cinematic directives for use in filming a story and Zuzen will produce a video file that reflects their specifications. This forgoes the usual learning curve associated with typical machinima or cinematic content creation tools. This paper describes the Zuzen framework and details its implementation.

Categories and Subject Descriptors
I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems – *games*

General Terms
Algorithms

Keywords
Artificial Intelligence, Cinematic Camera Control, Machinima, Service-Oriented Architectures

1.INTRODUCTION
Cinematics created using game engines (known as *machinima*), are gaining significant ground as both products for consumption as entertainment as well as artifacts expressing their makers' creative designs. Popular web sites (e.g. machinima.com, gamevee.com) host a wide range of videos created from game engines as well as tutorials, forums and tool downloads targeted at machinima creators. While there is a significant interest in creating machinima, current machinima tools are very labor intensive and require that a user construct each scene manually. This is typically done by a group of human players each controlling one of the cinematic's virtual characters and at least one more player controlling a first-person view into the world that serves as a camera. The camera's view is recorded to create a collection of shots which are then edited together by a human director after recording has ended using an external video editing program like Movie Maker, iMovie or Final Cut.

A significant challenge for an intelligent camera control system would be to completely automate this process, taking as input a specification of a scene's actions in the form of a script and a list of camera control directives to be used to film the action, then producing as output the resulting video. The work we report here describes initial work on such a system, called Zuzen, implemented as a cloud-based service that uses the Unreal engine to create cinematic videos based on client specifications of story and camera activity.

Zuzen relies on the composition of several existing systems to aid in the creation of videos. Darshak[4] is used to generate the actual creation of shots. This system provides a pre-constructed set of camera directives that can be used inside a 3D environment containing dynamic objects. Zuzen provides a web interface to Darshak and adds syntax for more precise timing of camera and story actions. Additionally, third party tools are used to automatically render the execution of those actions to video. The system provides those videos for download to clients via an HTTP service.

2.RELATED WORK
Machinima, films created using game engines, began on the Quake II game engine. Today, they created using a variety of more modern game engines, such as Epic's Unreal engine and Valve's Source engine. Current tools provided by companies for creating machinima, such as the Unreal engine's Matinée Editor, require that the user frame each shot and script each actor's behavior manually. Generally, this requires navigating a 3D environment and placing actors and individual cameras. Additionally, the user must script actor behavior in whatever language or interface the game engine or tool provides. This means that in order to create films using a game engine, a user must first learn how to use the engine's level editing tools. This is a hurdle for amateur film makers who might have extensive knowledge about cinematic concepts but may lack experience with game engine tools. This limitation has led to the creation of tools aimed at making the creation process more accessible.

Other more specialized tools for Machinima generation exist. iClone provides tools for creating customized actor models and manipulating those models on screen in a puppet-like fashion at a very low level of control (i.e., joint-level movement) during scenes. While such tools provide a full range of expressivity for the user, such fine control of actor movement may not be desired by all users. This method does not lend itself to rapid prototyping or "what if" pre-visualization sometimes used in film making. Additionally, like tools built into game engines, these tools also require that a user learn a specialized, possibly unfamiliar tool set. Finally, because these tools are installed and run locally on the users' machine, their effective use requires that users' systems

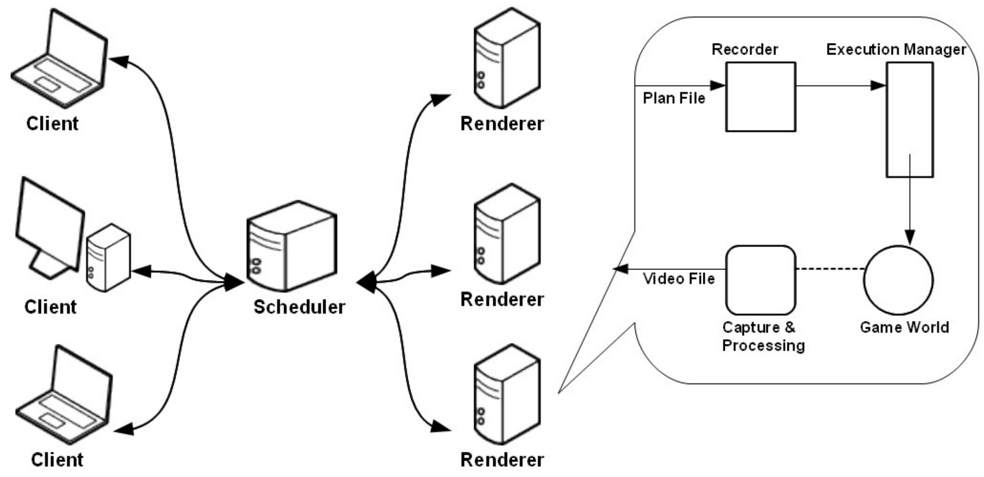

Figure 1: Zuzen Framework

have the graphics hardware needed to render the polygonal models used by the tool.

Recently, there has been work in automatic camera control systems intended to assist users in creating machinima. Drucker[3] describes automated camera control in a 3D environment using a task-centric approach. Bares[2] describes a set of camera directions to use in a 3D environment, "Front", "Front-Right", "Right", "Back-Right", "Back", "Back-Left", "Left", and "Front-Left". This eight direction compass rose approach allows camera shots to be more easily discretized. These discrete directional shots can more easily be dealt with by other systems, such as AI planners.

Additionally, Jhala[4] constructed a camera control system, named Darshak, that automates the control of the camera in the Unreal Tournament 2004 game engine. This system took in a set of story actions and and a set of camera directives that describe constraints on how the story actions were to be filmed. Those directives were linked to a library of pre-defined camera actions within the engine that exploit existing cinematic idioms in order to achieve a more “film-like” experience. Darshak resolved blocking and location constraints in real time and ran locally on a machine that needed to have the game engine and the camera system installed.

The automated control frameworks mentioned above allow a user to delegate some degree of control of the camera to a computer system. This is desirable for users who do not wish to specify exact camera directions.

Cambot[7] is a intelligent system for creating machinima from a script-like representation. Here, scripts are divided into a number of scenes, with each scene containing at least one beat. Additionally, Cambot supports a number of 3D environments or sets, each annotated with high-level 3D information relevant to Cambot's control needs. For each scene specified in a script, Cambot searches for appropriate selection of shot parameters by iterating through all possible sets of blockings for the given set that satisfies that scene and chooses the best one using a heuristic estimate.

In contrast to systems that require each user to provide computing or data access capabilities sufficient for all their application needs, service-oriented architectures provide users with access to high end computation via wed-based interfaces. For example, service oriented AI architectures have been used to allow planning frameworks to be used remotely over a web connection. Users can then construct partial plans locally and send them to a remote server for completion. This means that a user does not need to run any planning agents on their local machine and emphasizes a black box approach. The Zocalo[8] framework is a set of planning agents that are accessible over a network interface. Communication with Zocalo is achieved via XML-based plan files. These files specify a set of actions in a STRIPS-like format [10], listing preconditions and postconditions for each action. At the heart of the Zocalo framework is Crossbow, a causal link planner based on the DPOCL planning algorithm[9]. The system is able to compute plans based on actions that take place in an execution environment such as a game engine. Zuzen was modeled on this paradigm and thus can interface with parts of the Zocalo framework.

3.SYSTEM DESCRIPTION

Zuzen is a system for generating video files from a script-like plan file. It operates as a service and can take input from and send output to a variety of different applications. Communication with the system is done via HTTP. Zuzen is composed of two different internal modules that communicate with several third-party applications. The first of these modules listens for and processes incoming plan files. Once information about the actions contained in the plan file is extracted, this information is used to construct a set of parameterized function calls -- called action classes – that correspond to the relevant character and camera actions that can be executed in the game environment. The execution of these actions is managed by an Execution Manager, written in native Unreal code. The execution of these actions is recorded using screen recording software and the resulting video is stored and is available for the client to download over a web connection. These components are described in more detail below.

The first component is a C# module that listens for incoming socket connections and parses incoming plan files and uses this information to construct concrete Action Classes. The second component is an UnrealScript module that manages the execution of actions inside the game engine. The third and final component is a video capture module consisting of a screen recording program and a video compression utility. These modules work together in order to take in a plan file from the user and return a video file of the execution. This video can then be reviewed by the user and the action plan tweaked until the desired final product is produced.

The C# module, called the Recorder, listens for incoming network connections from clients. Once a connection between Zuzen and a client is established, Zuzen requests a plan file from the client. This plan file contains a list of actions that are to be executed for a set of scenes, as well as information about the timing between the actions in the scenes. This file is then parsed to extract this information about the actions.

In order for the actions to be executed within the game environment, they must be converted into native Unreal code. These pieces of code are called Action Classes. Each Action Class contains the code that will cause the action to occur within the game environment. These constructed action classes are then passed to the Execution Manager for further processing. A list of these actions is provided in Section 3.2.

The Execution Manager controls the execution of actions within the game environment. The Execution Manager is written in native Unreal code. This allows for the precise timing of actions that would not be achieved if execution timing was managed externally. Zuzen's Execution Manager can control the timing and execution of two types of actions, Story Actions and Camera Actions. At an abstract level, there is no difference in how these actions are specified. However, they control different things in the 3D environment and are therefore subject to different restrictions. Story actions control actions such as the movement and speech of characters and other events that occur in the environment. Any number of story actions can occur simultaneously in the world. Camera actions control the movement and direction of the camera. Zuzen can only record one perspective in the environment at a given moment, therefore, only one camera action can execute at a time.

Three queues are used to keep track of actions: the Pending Action Queue, which contains all of the actions that are waiting to execute; the Executing Actions Queue, which contains all of the actions which have started executing but have not yet finished; and the Executed Action Queue, which contains all of the actions that have finished executing. Initially, all actions are in the Pending Actions Queue. Each game “tick” the Execution Manager iterates through all of the actions in the Pending Action Queue and checks if they are ready to start executing. If they are, the Execution Manager starts the action and moves it to the Executing Actions Queue. Actions in the Executing Actions Queue remain there until they have finished executing, at which point they are moved to the Executed Actions Queue.

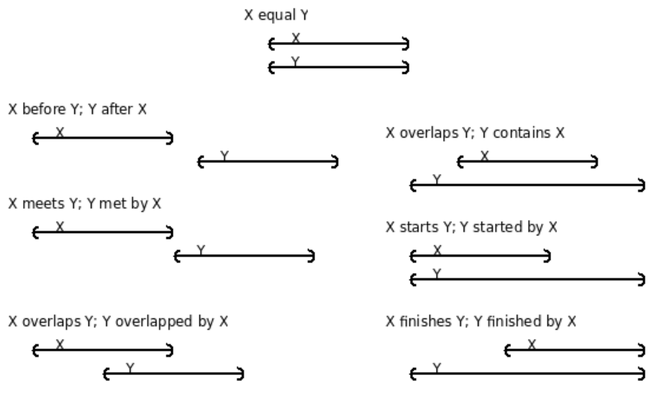
X equal Y
X
Y
X before Y; Y after X
X
Y
X overlaps Y; Y contains X
X
Y
X meets Y; Y met by X
X
Y
X starts Y; Y started by X
X
Y
X overlaps Y; Y overlapped by X
X
Y
X finishes Y; Y finished by X
X
Y

Figure 2: Temporal Intervals

In order to render the execution of actions to video, third-party applications are used. The execution of actions is recorded to video using the FRAPS application. This application records the computer's video buffer to a file on a frame by frame basis. Because this data is not compressed, the resulting video file is very large and cannot be easily viewed with common video players. Because of this, the file is encoded to a more common format using VirtualDub. This allows the final video file to be transferred over a network connection more quickly. Multiple scenes sent by the client are concatenated using AviSynth. Alternately, Flash previews of individual scenes can be generated using FFmpeg. This allows a user to fine tune a scene iteratively.
Because the Recorder module captures video directly from a single machine's video output, only one Recorder at a time can be executing on a given machine. However, Zuzen is capable of managing multiple rendering servers running the Recorder application. To provide this functionality, a separate server process running a Scheduler program is used. All incoming cinematic plans are initially sent to this schedule server. This Scheduler maintains a list of render servers that are currently idle. Whenever a plan is received, the Scheduler forwards it to one of the available render servers and marks that server as busy. Once rendering is complete, it returns the video file to the appropriate client and remarks the relevant Renderer as idle.

3.1 Temporal Language

Allen[1] describes thirteen types of relationships between two temporal intervals. These are: "before", "after", "during", "contains", "overlaps", "overlapped by", "meets", "met by", "starts", "started by", "finishes", and "finished by". A representation of these temporal relationships are shown in Figure 2. Being that these represent all of the possible temporal relationships between two intervals, any relation model between actions should attempt to be able to express all or as many of the thirteen relationships as possible.

In order to describe the temporal relationships between actions, a formal language is used. Relational operators in this language map to Allen's temporal relations. Actions within this language are described by their type and by any additional parameters that that action may require. Taken together, the set of actions and relations described by this language form a specification on a directed acyclic graph. As will be discussed, Zuzen currently implements only a subset of Allen's relationships.

Each action in the framework has a set of preconditions, a set of constraints, and a set of post conditions, thereby following a STRIPS-like approach. In the Zuzen framework, preconditions

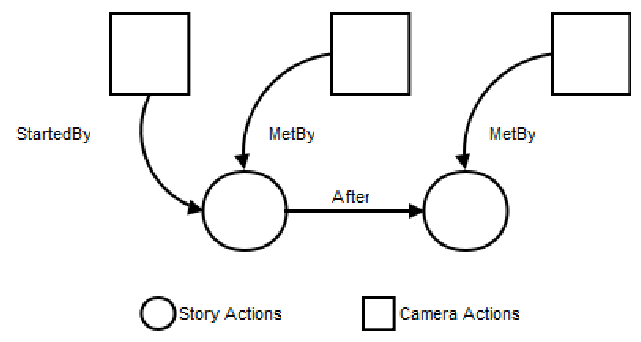

Figure 3: Zuzen DAG Structure

primarily concern timing; whether or not the actions predecessors have begun executing and whether sufficient time has passed between the actions. Constraints are used to ensure that actions are not used with inappropriate objects. For example, the Speak action can only be used with objects that are capable of speech. Postconditions are things that occur in the game environment in the course of an action executing. In order for an action to be considered successfully executed, all of its post conditions must be true. For example, for a MoveTo action to be considered successfully executed, the specified actor must have arrived at his or her destination.

In order to control the timing of actions in the game world, a relational model between actions must be specified. Specifically, it is necessary to specify in what order actions should be executed, as well as how actions overlap.

Additionally, because the actions used by the Zuzen are described in terms similar to those used in traditional planning frameworks, it would be useful if a similar relational model was chosen as this would allow planning systems to fill in gaps in incomplete plans or to suggest alternate plans.

Because of these considerations, the following model was used to describe the relationship between actions. Any individual action can have any number of predecessor actions and any number of successor actions. Predecessors of an action are actions that must start before that action while successors of an action are actions that must start after that action. This relationship between actions forms a Directed Acyclic Graph (DAG), a structure that is common in planning engines. Additionally, an action may be offset from one of its predecessor by a number of seconds. This offset may either be from the start of a predecessor of from the end of a predecessor. This restriction is necessary because actions occur in real time in the game environment.

While it is possible to know when an action starts (when it is moved to the Executing Actions Queue) and when an action is complete (when all of its postconditions are true), it is not possible for Zuzen code to know the exact time when an action's code ends its execution. This is because a) the underlying game engine does not provide system-level calls that can track thread/process-level events like function completion and b) no API is provided to estimate run-time of a specific function call. The execution times of story actions are of variable length within the game world. For example, the time needed for a MoveTo action will vary depending on the distance between the action's start and destination and the speed of the actor performing the action.

This limited knowledge about the execution state of an action means that we can specify *start point to start point* and *end point to start point* relationships between actions, but not *end point to end point* or *start point to end point* relationships between actions. In other words, we cannot specify that one action end at the same time another action ends.

Because each action has a duration and can be offset from other actions and each cinematic plan represents a total ordering of actions, the Relational Graph of camera actions and story actions may also be thought of as two parallel time lines. One time line mapping to story actions and the other mapping to camera actions.

3.2 Action Library

A pre-made library of character and camera actions is used to assist in the rendering of the scenes. The following actions are currently provided:

MoveTo *Actor Destination*
This action is used to have the specified actor walk to the specified location. Destination must be a named pathnode.

Speak *Actor Hearer Text*
This action causes the specified Actor to face the Hearer and the string Text to appear on the screen as a subtitle.

Jump *Actor Direction*
Causes the specified actor to jump in the specified direction.

Crouch *Actor*
Causes the specified actor to crouch. The actor will continue to crouch until an Uncrouch action is specified for that actor.

PerformEmote *Actor Emote*
The specified actor will perform a preprogrammed animation (e.g., waving hands, pointing hands, wiping sweat from the character's brow). Animations come from a library built into each character model.

LookAt *ObjectOfAttention Direction Distance [Duration]*
Causes the camera to look at the specified object from the given Direction, from the specified Distance. Optionally, a duration may be passed to the action.

Track *ObjectOfAttention Direction Distance [Duration]*
Causes the camera to look at the specified object from the given Direction, from the specified Distance. If the ObjectOfAttention moves, the camera will follow along. Optionally, a duration may be passed to the action.

Pan *ObjectOfAttention1, ObjectOfAttention2, Distance*
Causes the camera to pan from the first object to the second object from the specified distance.

Dolly *ObjectOfAttention, Distance [Duration]*
Causes the camera to dolly along with the ObjectOfAttention, following it if it moves. Optionally, a duration for the may be specified.

EstablishingShot *Shot*
Invokes one of the pre-defined establishing shots built into the level.

OverTheShoulderShot *Shoulder Target Distance Direction*
Cause the camera to position over the shoulder of one actor looking at the specified target in the specified direction and at the specified distance from the source actor's shoulder.

CameraEffect *Effect*
Causes one of a number of camera overlays to appear on screen. Possible effects are: Splatter, Fade In, Fade Out, Cut to Black.

AddProp *Prop Location*
Adds an inanimate object to the environment at the specified location.

TimeDilation *Scale*
Alters the passage of time in the environment by the given scale.

3.3 Setup Information

In order to render a given script, client systems must provide Zuzen a small amount of additional information beyond the set of actions to be executed. First, the level to be used for the filming of action must be specified, along with the location within the level where each scene will take place. These levels and settings are selected from a library of levels created for the specific game engine being used.

Additionally the name, model, and location for each actor in the scene must be specified. Actor models also come from a library of character models provided by the engine. Locations in the environment are named pathnodes that are placed in each level by hand by Zuzen designers.

3.4 Environments

A library of settings is used to aid in the construction of films. These settings provide a set of stock actors and a 3D environment for the actors to interact in. For each setting, a list of actor meshes and a set of top down maps showing the location of the named path nodes is provided. These maps are stored on the server as images and available over a HTTP connection. Additionally, a set of premade establishing shots exist within each level. Any setting in the system distinguishes location by using the naming conventions described below.

New levels may be added to this library using Unreal's built in level editor. These levels are created as normal and annotated pathnodes and establishing shots are added to the 3D environment. Hungarian notation is used to name these pathnodes: they are labeled with a “loc_” prefix and a short description of where it is located in the environment. For example, a pathnode near a desk in a library might be labeled “loc_LibraryDesk”. Similarly, establishing shots are prefixed with “es_” and a short description of what they display: a shot of the front of a restaurant might be labeled “es_RestaurantFront”.

Currently a setting based on American Western films is provided, known as WestWorld. This setting provides two separate stereotypical Western towns, one modeled on a Mexican villa and one modeled on a frontier town, and eight stock characters, including both male and female characters, to choose from. Each of these provided towns have a variety of visually different buildings.

3.5 Action Classes

Below are two examples of the UnrealScript code necessary to cause actions in the game world. A common method is used to parse incoming information from a plan file. This can be seen belows in the *bindings.getConstant(...)* method.

The first action presented is a MoveTo. This action causes an actor to move from its current location to a specified location. In this case, the game engine provides some assistance with this. The variable 'MoveTarget' is built into the engine and a unique instance of it is kept for each actor. Setting it causes the actor to move toward that location automatically.

```
TargetParam = bindings.getConstant("target");

newMoveTarget = FindWorldObject(TargetParam);

if (newMoveTarget == none) {
    return false;
}

MoveTarget = newMoveTarget;

return true;
```

Next we will examine a camera action. This is a simple LookAt action which will cause the camera to change it's focus to a designated target. Additional parameters allow the user to set view distance and view angle. First, we will examine the code to set the view target. Here the named object is found in the game world and set as the 'ObjectOfAttention' for the in game camera.

```
TargetParam = bindings.getConstant("target");

TargetObject = FindWorldObject(TargetParam);
```

```
if (TargetObject == none) {
    return false;
}

if(TargetObject.IsA('Controller')){
   TargetObject = Controller(TargetObject).Pawn;
}

ObjectOfAttention = TargetObject;
```

Next, we parse the shot distance from the list of parameters. Shot distance can only be set in discrete intervals. These intervals are mapped to an enum stored elsewhere in the class.

```
ShotTypeParam =
   bindings.getConstant("shotType");

switch(shotTypeParam) {
     case "SHOT_VERY_CLOSE":
          shotType = SHOT_VERY_CLOSE;
          break;
     case "SHOT_CLOSE":
          shotType = SHOT_CLOSE;
          break;
     case "SHOT_MEDIUM":
          shotType = SHOT_MEDIUM;
          break;
     case "SHOT_LONG":
          shotType = SHOT_LONG;
          break;
     case "SHOT_VERY_LONG":
          shotType = SHOT_VERY_LONG;
          break;
     default:
          shotType = SHOT_MEDIUM;
          break;
}
```

Finally, we parse the view angle, again this values is limited to a discrete set of choices. Here the choices correspond to the one of eight cardinal directions, relative to the facing of the actor being observed.

```
DirectionParam =
   bindings.getConstant("direction");

ViewAngle = rot(0,0,0);

switch (DirectionParam) {
     case "Front":
         break;
     case "FrontRight":
         ViewAngle.Yaw = 8192;
         break;
     case "Right":
         ViewAngle.Yaw = 16384;
         break;
     case "BackRight":
         ViewAngle.Yaw = 24576;
         break;
     case "Back":
         ViewAngle.Yaw = 32768;
         break;
     case "BackLeft":
         ViewAngle.Yaw = 40960;
         break;
     case "Left":
         ViewAngle.Yaw = 49152;
         break;
     case "FrontLeft":
         ViewAngle.Yaw = 57344;
         break;
     default:
         break;
}
```

4.CURRENT APPLICATIONS

Zuzen is designed as a cloud-based service. As such, it does not have an interface other than manually writing the required plan files. However, other applications can interact with it in a meaningful way. Two sample applications are described below. The first, Darshak is a planning framework designed to work with cinematic plans, like those used by Zuzen. The second, Longboard, is a tablet-based graphical front end that can interface to Zuzen in order to render its videos.

4.1 Darshak

Darshak is an intelligent camera planning system. A plan containing a sequence of story actions is be passed to the planning agent. Using this information, Darshak will create a plan containing a sequence of camera actions displaying the given events using knowledge about cinematic idioms. This completed discourse can then be processed by the Zuzen framework in order to generate a completed video. Darshak seeks to satisfy three main requirements in order to generate a coherent cinematic discourse. First, the director seeks to extract the “salient elements” from the given discourse. Second, these elements are organized into a rhetorical structure that will allow for a coherent telling of a story containing the events. Third, camera shots are chosen to depict the events is the story such that a set of cinematic constraints are satisfied.

Darshak uses a variation of the DPOCL planning algorithm called DPOCL-T, adding temporal constraints to the planning domain. Temporal constraints are used by Darshak to maintain explicit timing relationships between temporal variables describing the start and end times of actions. Only two types of temporal constraints are used by Darshak: before and equals. As a result, actions are represented by arranging them along a directed graph with directionality representing the passage of time.

Darshak also includes a module for executing the plans it produced inside a 3D virtual environment. This allowed a user to produce and tune plans on a single workstation. Partial plans may also be sent to and from Darshak over a TCP connection. This means that Zuzen and Darshak may either be run of the same server or on different servers.

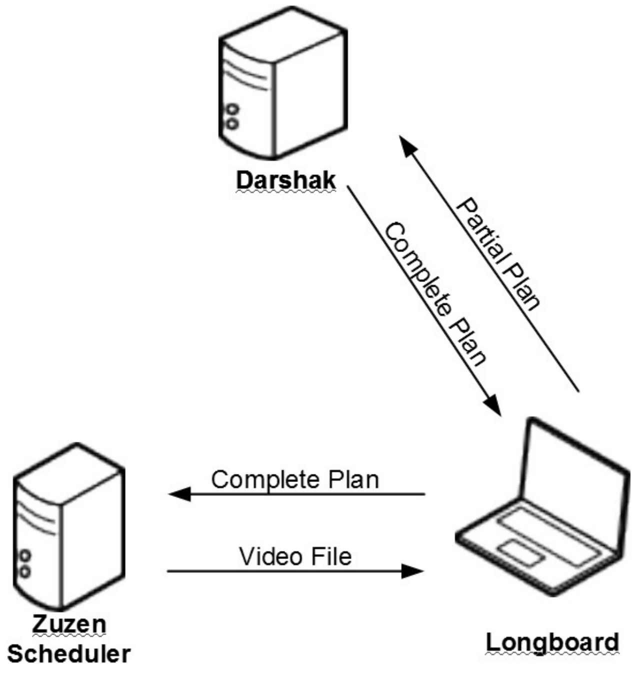

Figure 4: Longboard Workflow

4.2 Longboard

Longboard[7] is a sketch-based storyboarding tool that serves as a front end for Zuzen. In Longboard, individual storyboard frames are created by sketching them out on a TabletPC. These sketches are supplemented by annotations for each frame which give additional cinematic directions, such as actor destinations, actor dialog and camera timing. The information provided by the user in the TabletPC client application is converted to XML plan files representing story and camera actions in Zuzen's required format. Because actions must occur between storyboard frames in order to form a coherent time line, these plan files are incomplete and are missing pieces of information that are necessary to render the scene. This missing information can include additional camera shots or actor movement.

To flesh out the needed action details, these incomplete files are sent to an external planner, Darshak. Darshak fills in any information that is missing from the plan file by adding additional actions. The newly completed plan file contains all of the information necessary to render the scene. This file is sent back to the TabletPC. The file is then sent from the tablet client to Zuzen and rendered to video using the methods described above. The rendered video is then sent back to the the TabletPC for review.

Longboard emphasizes an iterative approach to building a a video file. If a user is unsatisfied with the resulting video, he or she can replace existing storyboard frames with new ones or add additional frames until the resulting cinematic is acceptable.

5.DISCUSSION

One way we evaluate Zuzen is to characterize the system's ability to model the temporal relationships between story and camera actions described by Allen and mentioned above. Using the system of predecessors and offsets, a subset of Allen's temporal relationships can be easily defined. Below, the definable relationships are shown and we describe how they are represented using the relational model described above.

```
X Before Y: X successor of Y;
            Y offset from X by i
X After Y:  Y successor of X;
            X offset from Y by i
X Meets Y:  X successor of Y;
            Y offset from X by 0
X Met by Y: Y successor of X;
            X offset from Y by 0
```

```
X Equal Y:
  X and Y equal length;
  X successor of Y;
  Y offset from start of X by 0
X Overlaps Y:
  X successor of Y;
  Y offset from start of X by i
X Overlapped by Y:
  Y successor of X;
  X offset from start of Y by i
X During Y:
  X shorter than Y;
  Y successor of X;
  X offset from start of Y by i
```

```
X Contains Y:
  Y shorter than X;
  X successor of Y;
  Y offset from start of X by i


X Starts Y:
  Y longer than X;
  X successor of Y;
  Y offset from start of X by 0
X Started by Y:
  X longer than Y;
  Y successor of X;
   X offset from start of Y by 0


X Finishes Y:
   Y longer than X;
   X successor of Y;
   Y offset from start of X by i
X Finished by Y:
   X longer than Y;
   Y successor of X;
   X offset from start of Y by i
```

While the definition of these four relationships is straightforward, definitions for the other nine temporal relationships are not so easily provided. This limitation arises from the inability of the system to specify the duration or end time point of executing actions. The remaining relationships require knowledge about both the start and end points of each action. Nevertheless, we have found in practice that the four relationships modeled within Zuzen are sufficient to represent a wide class of cinematics. Our future work will either extend the underlying scripting environment for Zuzen to provide duration information about executing actions or will or port Zuzen to work with a game engine that provides this functionality natively.

A second limitation with Zuzen's implementation is the requirement that spatial details (e.g., the placement of pathnodes, the creation of images that show the pathnodes within a given level's sets) must be created manually by Zuzen developers. Currently, each pathnode in an environment must be placed and named manually inside the Unreal level editor, UnrealEd. This is a very time consuming process requiring the developer to navigate around the 3D environment. Once the process is complete, a top down map of some kind must be created and annotated so that knowledge of the spacial relationships between nodes in the level can be maintained. This knowledge is necessary to ensure that actors move where they are supposed to. In order to get top-down maps of the interior or buildings, it is often necessary to remove pieces of the level geometry, either the roof of a building or, in some cases, the entire upper floor. Automating some or all of this process will significantly reduce the amount of work required to prepare a new setting.

6.ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation (awards #0450102 and #0915598). The authors wish to thank the Lawdogs Unreal development team for use of their character models in Darshak and in Zuzen.

7.REFERENCES

[1] Allen, J. F., Maintaining knowledge about temporal intervals in *Communications of the ACM* 26:832-843, 1973.

[2] Bares, W and Lester, J., Cinematographic User Models for Automated Realtime Camera Control in Dynamic 3D Environments in the *Proceedings of the Sixth International Conference on User Modeling,*. 1997

[3] Drucker S. and Zelter D., Intelligent Camera Control in a Virtual Environment in the *Proceedings of Graphics Interface*, 1994.

[4] Jhala, A. and Young, R. M., Representational Requirements for a Plan Based Approach to Automated Camera Control in the *Second Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2006.

[5] Young, R. M., Story and discourse: A bipartite model of narrative generation in virtual worlds in *Interaction Studies* 8:177-208*,* 2006.

[6] Elson D. K. and Riedl, M. O., A Lightweight Intelligent Virtual Cinematography System for Machinima Generation in the *Proceedings of the 3rd Annual Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2007.

[7] Jhala, A., Rawls, C. and Young, R. M., Longboard: A Sketch Based Intelligent Storyboarding Tool for Creating Machinima in the *Proceedings of the Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2008.

[8] Young, R. M., An Overview of the Mimesis Architecture: Integrating Intelligent Narrative Control into an Existing Gaming Environment in *The Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2001.

[9] Young, R. M. and D. Moore, J. D. DPOCL: A Principled Approach to Discourse Planning, in the *Proceedings of the Seventh International Workshop on Text Generation,* 1994.

[10] Fikes, R. and Nilsson, N., STRIPS: a new approach to the application of theorem proving to problem solving in *Artificial Intelligence* 2:189-208, 1971.