Image-space Constraints for Controlling Camera Interpolation

Ross Sowell, Tom Erez, Emily Feder, Cindy Grimm
Washington University in St. Louis *

Jianqi Xing
Illinois Institute of Technology †

Leon Barrett
University of California, Berkeley ‡

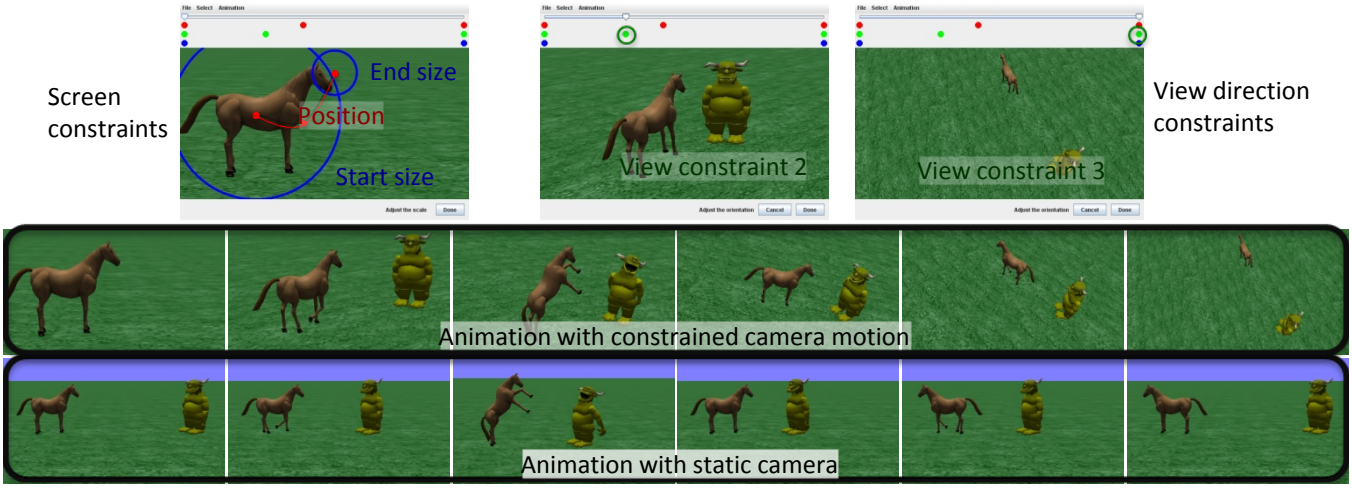

Figure 1: ***Specifying a camera motion using in-screen constratins. Top row:*** *(Left) The user specifies the path the horse should take using the spline curve (red dots). They use the blue circles to specify that the horse starts out big and gets small. (Middle, Right) The user also specifies the desired viewing direction at two other points in time.* ***Middle row:*** *The original animation viewed with a camera motion that meets the user's constraints.* ***Bottom row:*** *The original animation sequence viewed with a static camera.*

Abstract

This paper presents a novel interface for using image-space constraints to control camera interpolation in animation sequences. Traditional camera control can be challenging because the user must envision how the camera should be positioned in order to place the objects on the image plane the way they want. While it is fairly simple to place an object in the center of the scene and rotate the camera around it, more complex camera motions that involve both view direction change and motion of the object across the screen can be very difficult to envision and implement with smooth motion. In contrast, we provide a simple interface that allows the user to directly draw out the trajectory, size, and orientation of an object on the screen, while the system automatically solves for a sequence of cameras that satisfies those constraints. Unlike previous image-space constraint approaches, we use a constraint vocabulary which is both easy to use *and* produces more stable solutions.

CR Categories: I.3.6 [Methodology and Techniques]: Interaction techniques; I.3.7 [Three-Dimensional Graphics and Realism]: Animation

Keywords: camera control, keyframing, image-space constraints

1 Introduction

Camera keyframing is an integral part of the animation making process. The animator places the camera in a sequence of "key" positions, and the computer produces a set of intermediate camera locations that interpolates between these keyframes. This approach has several benefits. First, the animator only has to specify a small number of camera positions. Second, the interpolation produces smoother motions than hand-placement does. Third, the timing and number of output frames can be adjusted independently of the keyframes themselves.

Camera keyframing traditionally treats the camera as just another 3D object in the scene, with intermediate frames produced by interpolating the position and focal point of the camera in space. Unlike a 3D object, however, the camera's role is to *project* the 3D scene into 2D. The animator indirectly controls the projection — how objects in the scene are placed in the 2D image — by adjusting the camera parameters for each keyframe. Automatic 3D camera interpolation adds yet another layer of indirection. The net effect is that the animator must solve a complicated inverse problem in order to move objects across the 2D scene in the desired manner. Figure 7 shows an example where the animator wanted the table to move down and across the scene while moving the view to the

*e-mail: {rsowell, etom, eafeder, cmg}@wustl.edu
†e-mail: jxing1@iit.edu
‡e-mail: lbarrett@eecs.berkeley.edu

top of the table. Traditional interpolation rotates the table out of the view. Correcting this using traditional approaches requires the addition of a substantial number of keyframes which in turn reduce the smoothness of the motion.

Image-space constraints [Blinn 1988; Gleicher and Witkin 1992] were introduced as a solution to the inverse problem. The animator specifies the desired image-space constraints (this object should be here) and the system solves the inverse problem to determine the correct camera path. Unfortunately, except for a few types of animations (flying around an object, panning across a scene), this approach is unstable and difficult to control [Barrett and Grimm 2006]. There are several reasons for this, but one of the primary problems is that there are multiple ways to move the camera in order to account for changes to the image-space constraints. If the image-space constraints are relatively simple (translation across the image) the system behaves well, but for more complicated constraints, such as a rotation plus a translation, the resulting camera paths are jerky, or may pass through un-intuitive camera positions.

We expand on the spirit of Gleicher and Witkin's work to provide a more usable interface, a direct solve in the case of a single object, and a robust solver when the user wishes to control multiple objects. Our interface has the notion of a keyframe, but instead of specifying an entire set of camera parameters, the animator specifies one (or more) image-space constraints — the object must be here, it must be this size, and this is the view direction. To determine where the camera is for any intermediate frame, the system interpolates these constraints then solves for a camera that meets them, while producing a smooth camera path. The interpolations can be visualized, and edited, in the image plane in order to control what happens *between* the keyframes as well.

Traditional camera interpolation works well for a certain class of animations, such as following an object or panning over a scene. Our interface can, of course, be used to make these traditional camera motions, but its real strength is that it makes it easier to specify camera motions where the traditional camera actions (look at that object, pan across the scene, etc.) do not apply.

Contributions: Our first contribution is an in-screen camera animation interface (Section 3). This interface has been incorporated into Alice [Alice 2010]. Second, we suggest a more natural set of constraints or controls. These controls have the added benefit that they do not result in as many ambiguities as pure point or line constraints do. Third, for the case of constraining a single object, we can directly build a camera motion that meets those constraints (Section 4). Forth, we present a general solver, based on a local trajectory optimization approach called Differential Dynamic Programming (DDP) [Jacobson and Mayne 1970] to solve for a camera motion path (Section 5). The DDP approach tends to produce physically-plausable motions by minimizing acceleration energy.

2 Related Work

Traditional 3D key framing requires interpolation of position, orientation, and the intrinsic camera parameters. The position and intrinsic camera parameters are easily interpolated using linear weights. Shoemake [Shoemake 1985] introduced quaternion interpolation for rotations. Barr *et. al.* [Barr et al. 1992] expanded on this idea to produce spline-style interpolations between two quaternions. Kem *et. al.* [Kim et al. 1995] describe how to blend between more than two quaternions, essentially treating each quaternion as a control point on a C^2 spline. Alexa [Alexa 2002] and Hofer and Pottman [Hofer and Pottmann 2004] both offer general-purpose methods for interpolating rigid body motions, and Hawkins and Grimm [Hawkins and Grimm 2007] showed that Alexa's method can be adapted to direct interpolation of keyframes, even for sequences containing both rotation and center of projection changes. However, none of these methods give the user any control over the intermediate frames that are generated automatically.

Various systems have been proposed for semi-automatic or automatic control of the camera. Drucker and Zeltzer [Drucker and Zeltzer 1995] proposed a system, based on cinematic rules, for determining the sequence of camera shots in a virtual conversation. The individual camera shots had pre-defined locations for the camera; the system largely focused on rules for switching between them. He et. al. [wei He et al. 1996] expanded on this idea, allowing the individual camera shots to alter the 3D scene slightly to better frame shots. Tomlinson et. al. [Tomlinson et al. 2000] allow the characters themselves, and the evolving story line, to play a role in creating the camera shots. They model the camera as a 3D object attached to the characters via a system of springs and dampers — although the placement of the camera for a particular shot-type is still pre-defined, the dynamics allow some control over the "emotional" content in the camera motion. All of these systems rely on having a small set of (possibly parameterized) pre-defined camera shots to use as building blocks — they do not allow general camera placement. We are interested in creating novel camera paths from with arbitrary object placement and view direction.

Blinn [Blinn 1988] pioneered the use of image constraints for camera fly-bys. His approach used the "look at" camera model and constraints that were natural in that model (*e.g.*, center the view on that planet). Gleicher [Gleicher and Witkin 1992] presented a more general approach based on existing Inverse Kinematics solvers. The examples in the paper were primarily under-constrained and used an additional set of soft constraints to minimize changes to the camera from frame to frame. The experiments of Barrett and Grimm [Barrett and Grimm 2006] found that this solver tends to get stuck in local minimum, especially when the constraints are not satisfiable. The overall error also tends to increase over time, especially if, at any point in the sequence, the constraints are not satisfiable. We provide a more robust solver and a more controllable interface. We allow all the camera parameters (or a specified subset) to change, rather than just the position and orientation, and we allow an object's position, size, and orientation to be controlled separately, rather than using a single set of point or line constraints to handle all three attributes.

Computer vision researchers have developed many techniques for recovering camera parameters from projections of known points. Early work focused on planar [Zhang 1999] or conic [Song 1993; Wang et al. 2003] calibration patterns. More recently, linear and non-linear techniques for non-planar patterns have been developed [Triggs 1999; Reid et al. 2003; Nister 2003]. In all of these approaches the intrinsic parameters are assumed to be fixed across the sequence, and only pose parameters (rotation and translation) are recovered per frame. These techniques are primarily concerned with stability (with respect to small perturbations of the tracked points) and accuracy of projection on a frame-by-frame basis; they are not concerned with ensuring visually smooth camera paths. We also have explicit orientation constraint and size constraints, not just point constraints (and usually not sufficient point constraints for standard pose recovery). For these reasons we use a solver that can incorporate more general constraints and attempts to find smooth motions (Section 5).

3 User Interaction

The input to our interface is a 3D animated scene. We use Alice for most of the examples in this paper, but any 3D animation software could be used. The user first creates an animation in Alice, then

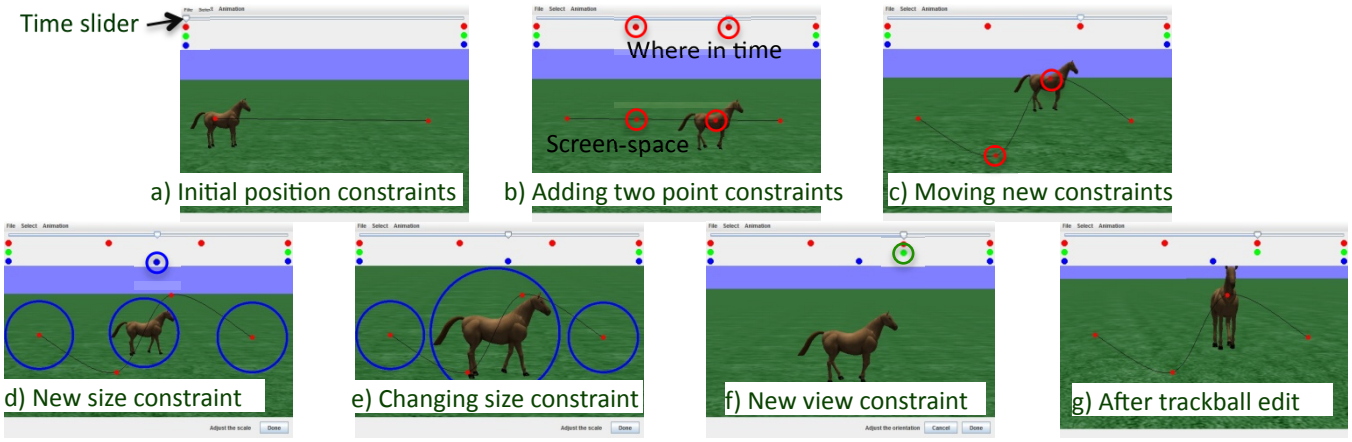

Figure 2: *(a) The initial trajectory for a horse walking left to right across the screen. (b) Two position constraints (red dots) are added to the trajectory. (c) The first constraint point is moved down, and the second up, to manipulate the path the horse will take on the screen. (d) A size constraint (blue dot) is added halfway through the animation. (e) Increasing the radius of the circle increases the size of the horse on the screen in the middle of the animation. (f) A view direction constraint (green dot) is added. (g) Using the trackball, the view direction is set to obtain a front facing view of the horse, causing the horse to rotate to face the viewer 3/4 of the way through the animation, after which it rotates back.*

brings it into our interface to create a camera motion. Our interface, shown in Figure 2, has a time slider at the top, and three rows (red, green blue) with dots to indicate the temporal location of the three types of constraints (position, size, orientation).

The user begins by selecting an object in the scene. The interface then shows two red dots, connected by a line, in the screen for the starting and ending projected screen position of the center of that object, using the default camera for that scene (see Figure 2(a)). (The two points may not be the same if the object moves in space during the animation.) The user can, at any time, "scrub" through the animation by moving the slider at the top of the interface window. At this point, the user can start adding in position, size, and orientation constraints, in any order. To add a constraint, they move the time slider to the desired temporal location, then select `Add ...` from the menu. New constraint values are always interpolated from the existing ones.

Position constraints: The position constraints are represented by 2D red dots, which are interpolated with a 2D Hermite spline. The user creates a new constraint by moving the time slider to the appropriate location in the animation and asking for a new point constraint (see Figure 2(b)). They can then grab this point and move it around the screen (see Figure 2(c)). If points are on top of each other (the user wants the object to remain in place on the screen) then the point that is closest to the currently selected time is the one that is selected.

Size constraints: As with the position constraints, the system automatically adds in a starting and ending size constraint, based on the size of the object in the default camera. These constraints are drawn as circles on the screen, representing the bounding sphere of the object projected onto the image plane. The user simply grabs the circle and makes it bigger (or smaller) (see Figure 2(e)). To create a new size constraint, the user moves the time slider to the desired location and asks for a new size constraint (see Figure 2(d)). The sizes are interpolated using a 1D Hermite spline.

View direction constraints: As with the previous constraints, the system automatically adds a starting and ending view direction. This constraint is a *3D* constraint, and is given by the desired view direction (stored as a quaternion). Unlike the previous two constraints, this constraint is not represented by any geometry in the image plane. Instead, when the user selects a view direction constraint to edit, the object moves to the center of the screen and the system enters a traditional trackball mode. The user interactively rotates the object to the desired viewpoint (see Figure 2(f,g)). When they are done, they click "done" and the view returns to normal. We use spherical linear interpolation (slerp) [Shoemake 1985] to interpolate the view direction constraints.

At all times the user can scrub the time slider from left to right and view their animation with the current camera path. As constraints are added and changed, the view automatically updates.

Using these simple controls, the user is able to specify the position, size, and orientation that the object should have on the screen over time. Note that the user does *not* have to specify all three constraints for every keyframe — each constraint type is interpolated independently. Given these constraints and a time value t, our system interpolates the constraints at t, then builds a camera that satisfies the resulting position, size, and view direction constraints.

In the following section we describe a direct-solve technique that can be used when constraining a single object. Because the degrees of freedom (two position, one size, three orientation) are fewer than the degrees of freedom of the camera, we can always find a camera that satisfies these constraints. We rely on the fact that the constraints change smoothly to produce a camera motion that is also smooth.

In Section 5, we discuss a general-purpose solver that can be used when the user wants to control more than one object at a time. In this case, there may not be a solution, so the solver looks for a smooth camera path that minimizes the constraint error.

4 Direct solve

In this section we describe our direct solve technique. Each of the constraint types in the previous section produces a continuous constraint parameterized by time. The point constraints are interpolated with a 2D Hermite curve, the size constraints with a 1D Hermite curve, and the orientation with spherical linear interpolation. Given a time value t, the direct solver produces a camera that meets those constraints for that time t. For simplicity's sake, the following dis-

cussion assumes that all of the values are for a given time t, and we drop the t. It should be noted, though, that every input is parameterized by time — moving the object changes the object's centroid, scaling it changes the bounding sphere, and positioning the default camera changes the distance.

The solver takes in the object's 3D centroid, P, its desired 2D location p, the radius of the bounding sphere R, the desired 2D radius r, and the desired rotation Q in the form of a quaternion. In addition, we have (from the original Alice camera) a default distance from the object to the camera D. (If more than one camera were specified, we interpolate D between those keyframes.)

The camera is defined in a three-step process. First, the camera is positioned so that it is looking down the z-axis at the point P, at a distance $||P - (0, 0, D)||$. Second, the camera is rotated around P to the desired view direction, Q. Third, using similar triangles, we calculate the zoom angle α required to project the bounding sphere of radius R to the circle r, where α' and r' are the initial zoom angle and the initial 2D radius (projected radius before scaling), respectively:

$$\alpha = \alpha' \frac{r'}{r} \tag{1}$$

Because extreme zoom angles can cause rendering issues, we clamp α between 10 and 80 degrees. To make up the remainder of the size change, we move the camera in (makes the object bigger) or out by scaling the current distance D':

$$D = D' \frac{r}{r'} \tag{2}$$

At this point, the object is centered in the scene, the appropriate size, and has the desired view direction. The last step is to translate the camera parallel to the image plane so that P is projected to the point p on the screen. Place the image plane in space centered at P, and place p on that image plane. The distance between these two points is the amount to move.

This solver correctly interpolates the position and size constraints, but is slightly off on the orientation constraint if the object is shifted out of the center of view. We have not found this to be a problem in practice.

5 Multiple objects

This section explains how to solve for a camera motion when there is more than one constrained object. Unlike the previous approach, we do not solve for a single camera in isolation, but for the entire camera sequence at once. This allows us to optimize for both the constraints *and* a smooth camera path. We use a solver based on Differential Dynamic Programming (DDP) [Jacobson and Mayne 1970], a technique from the reinforcement learning literature. DDP was shown to generate locally-optimal behavior even in the absence of a reference trajectory [Tassa et al. 2008], and since it approximates the value function only along a single trajectory, it does not incur an exponential computational cost in the degrees of freedom. Furthermore, DDP solves problems of continuous variables, and so required no special adaptation to address our problem.

To use DDP, we need to turn our constraints into a cost function. The smoothness of the camera motion is automatically handled by the way DDP frames the problem — it essentially treats the camera as a physical object that is moved by forces over time. DDP tries to minimize applying forces while meeting the constraints at each time step.

We first describe the camera representation we use, and then the cost constraints.

5.1 Camera representation

We use the Four-point camera model [Barrett and Grimm 2006] which is a geometrical representation of a camera comprised of a point and three vectors (see Figure 3(a)), for a total of twelve parameters. However, scaling all three vectors by the same amount results in the same image, so in effect there are only eleven parameters, as is the case in the traditional camera matrix [Michener and Carlbom 1980]. These two models are interchangeable (see [Barrett and Grimm 2006] for details of the conversion) – it's simply the *meaning* of the parameters that changes. The Four-point model was chosen here because it makes the specification of the image-space constraints geometrically meaningful, which in turn produces better-behaved cost functions. The parameters of the Four-point camera model are as follows:

1. O - the origin of the camera's coordinate system, or the pinhole of the camera.
2. $\vec{F}$ - the vector from O to the center of the film plane.
3. $\vec{U}$ - the film-plane vector in the x direction (right vector).
4. $\vec{V}$ - the film-plane vector in the y direction (up vector).

When solving for the intermediate frames, we allow the animator to specify the degrees of freedom (DOF) of the camera. By default, all of the parameters are free (11 effective DOF). Optionally, we can force $\vec{U}$ and $\vec{V}$ to be perpendicular by specifying $\vec{V}$ as a ninety degree rotation of $\vec{U}$ about $\vec{F}$, effectively eliminating skew (10 DOF). Forcing $|\vec{U}| = |\vec{V}|$ = aspect ratio eliminates aspect ratio scaling (9 DOF), and forcing $\vec{U}$, $\vec{V}$, and $\vec{F}$ to be mutually perpendicular eliminates center of projection changes. The latter is enforced by specifying $\vec{V}$ in terms of $\vec{U}$ as before, and then heavily penalizing values of $\vec{F}$ that are not perpendicular to $\vec{U}$ (effectively 7 DOF).

5.2 Cost Function

For simplicity, we define the cost function for a single frame, one object, and one constraint type. The full cost function is the sum over all frames and all objects and all constraints. Note that the user does not need to specify all three constraint types for each object, nor do they even need to specify the constraint for the entire time period. Variables are the same as in Section 4.

The cost function takes in the current camera ($C = O, \vec{F}, \vec{U}, \vec{V}$) and produces a measure of how well that camera satisfies the constraints.

Trajectory Cost: Given the object's 3D center P and the desired screen location $p \in [-1, 1] \times [-1, 1]$, measure how far the projected location is from the desired one. Normalize this by the size of the image plane. First project P onto the plane $O + \vec{F}, \vec{U}, \vec{V}$ to yield P' (see Figure 3(c). Then the cost is:

$$T(C) = \frac{||P' - (O + \vec{F} + p_x\vec{U} + p_y\vec{V})||}{||\vec{U}|| + ||\vec{V}||} \tag{3}$$

Scale Cost: Given the object's bounding sphere radius R and a desired screen size r, measure how much the projection of R onto the image plane differs from r. We use similar triangles to calculate

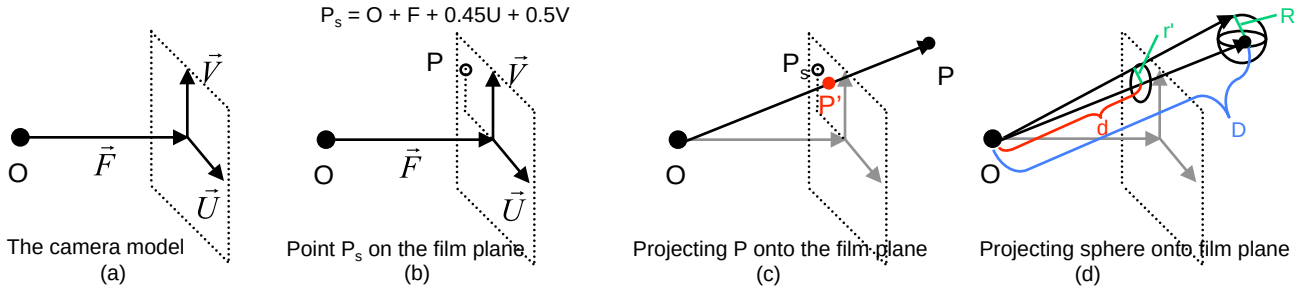

Figure 3: *(a) The Four-point camera model is constructed from an eye point, a vector to the film plane, and two vectors which span the film plane. (b) An image-space constraint p is converted to a 3D point P_s on the film plane. (c) Projecting P onto the film plane (P'). (d) For the scale constraint, the radius r of the projected circle is found by scaling the radius R of the bounding sphere by the ratio of the distance from the eye to P' to the distance from the eye to P.*

the difference. Let D be the distance from O to P, and let d be the distance from O to P' (P projected onto the image plane). By similar triangles, we have $r' = \frac{d}{D}R$, yielding:

$$S(C) = \frac{(r - r')^2}{||\vec{U}|| + ||\vec{V}||} \tag{4}$$

Orientation Cost: Let Q be the desired quaternion, and $Q(C)$ be the actual quaternion of the input camera:

$$R(C) = (1 + < Q, Q(C) >)/2 \tag{5}$$

All three cost terms are normalized so that they are (roughly) between zero and one, with zero being satisfied. Obviously, one constraint can be made stronger by weighting that cost term more.

We have verified that, in the case of a single constrained object, this solver meets all of the constraints. Although this general solver is mostly intended to be used with multiple objects, it can also be used with a single object when the user wishes to control the internal parameters of the camera (eg center of projection) through keyframing (Section 6.2).

Like all solvers, DDP performs best if the starting condition is close to the optimal solution. For the starting camera trajectory, we use our direct solver with one of the constrained objects. As a check on the solver, we have also initialized the solver with a variety of other starting conditions (traditional keyframe interpolation, linearly interpolating the four-point camera values) and it still converges, just much slower (3 minutes versus a few seconds).

The computational complexity of the DDP algorithm is linearly dependent on the number of interpolated frames [Liao and Shoemaker 1992]. We used a general-purpose implementation of DDP in MATLAB, and every camera sequence took a few seconds to compute on a standard desktop computer. While not yet fast enough for fully integrated user interaction, it is known that the DDP algorithm can be optimized for a specific domain and achieve dramatic performance improvement: for example, in [P. Abbeel and Ng 2007], DDP was brought to perform at a rate of more than 100Hz on a system of more than 20 dimensions by carefully optimizing the routine to the particular domain.

6 Results

In this section we provide several examples that illustrate the utility of our system. Complete animations of these examples are provided in the accompanying video.

6.1 Alice interface

Figure 5: *Recreating the sequence in Figure 1 using traditional keyframing tools in Alice. (a) The camera is moved about the scene and camera markers are dropped at the key positions. (b) The animation script with the "'moveAndOrientTo"' commands.*

Our direct solve system has been integrated into Alice. A user can load an existing Alice animation, and use our interface to edit the movements of the camera. In the example in Figure 1, the user begins with an animation of a horse meeting an ogre. The ogre roars, causing the horse to rear, and the horse turns and walks away. By specifying the trajectory of the horse with a simple spline curve, two size consraints, and adjusting the view direction at two points in time, the user is able to achieve an interesting camera motion. The camera starts zoomed in on the horse, with the ogre out of view. It then rotates behind the horse for the confrontation, and up to a zoomed-out, birds-eye-view as the horse walks away.

Creating such a camera motion with traditional tools is much more difficult. Alice provides traditional keyframing by allowing the user to adjust the camera parameters and save it as a camera marker (see Figure 5(a)). The user can then invoke a "'moveAndOrientTo"' method to force the camera to interpolate to any specified camera marker. Alice also provides other convenience methods such as "'moveTo"', "'orientTo"', "'lookAt"', etc (see Figure 5(b)). These tools are fine for simple pans and orientations, but are very diffcult to use for anything more complex, particularly for novice users. We made an effort to recreate Figure 1 using the traditional tools. Specifying and iteratively adjusting five keyframes still did not achieve the desired effect.

We ran a (very) informal user study where we asked people to make an animation, then create a camera motion both with our interface and Alice's camera editing tools (see Figure 6). Users were able to create camera motions using both systems. They commented, though, that our interface was more intuitive. Our users varied on

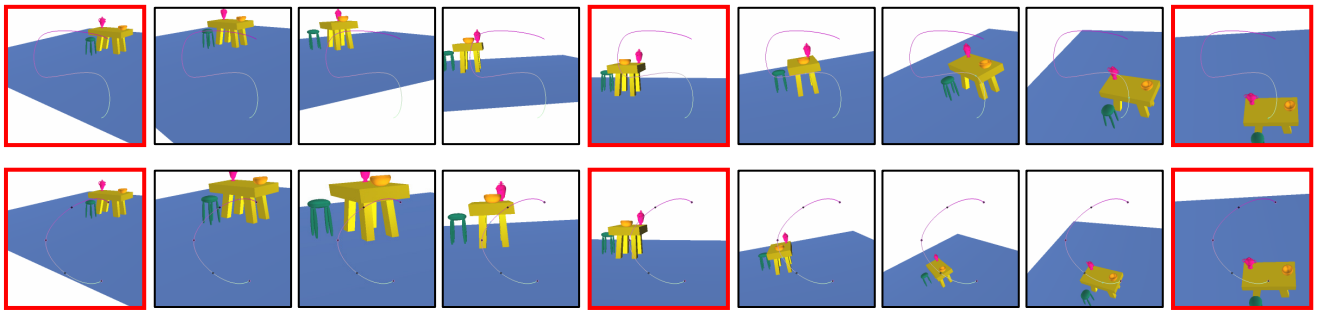

Figure 4: *Two sequences constructed from the same keyframes as in Figure 7 (3 key frames – highlighted in red, 120 total frames, 7 DOF camera model).* ***Top row:*** *The image-space trajectory has been modified so that the table will follow an "S" curve.* ***Bottom row:*** *The scale constraint has been modified so that the camera will zoom in and then back out during the first half of the sequence, and then zoom out and back in during the latter half.*

how they used the Alice camera controls; some just dropped cameras into the scene, others used the built in camera methods. The users who dropped cameras commented on how much easier it was to get something meaningful in our system versus this approach.

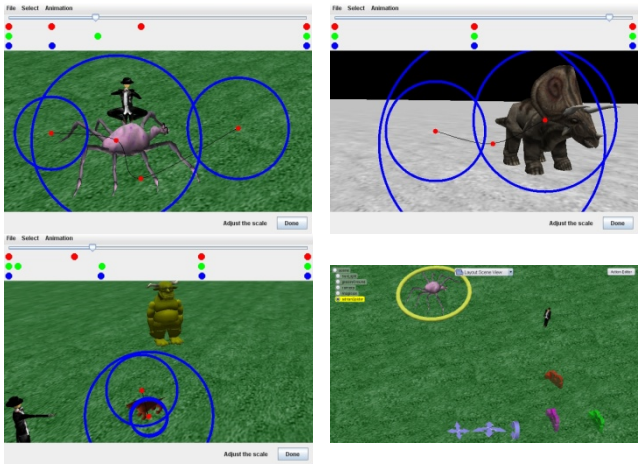

Figure 6: *Example scenes created by our users. Bottom right: A screen shot of editing the Alice camera.*

6.2 Traditional keyframing

Although our approach is primarily designed to be used interactively, traditional keyframing can also be used (see Figure 8). In this case, the user specifies two (or more) keyframes, and one (or more) objects. The system automatically generates each of the three constraints for each of the keyframes, and also interpolates the camera-to-object distance D (Section 4. Once the constraints are created, the user is free to edit them or add additional ones.

We use keyframing to demonstrate the well-known special effect where the camera dollies towards its subject while the lens zooms out, thus keeping the subject the same size in the frame throughout. The scene in Figure 8, top row, is a room full of tables, one of which has a bowl on top of it. The first keyframe shows this table in the upper left corner. Between the first keyframe and the second keyframe, the table is rotated and gets larger, while moving toward the center of the screen. Between the second and third keyframes, a dolly-zoom is executed, moving the camera toward the marked table while zooming out, creating the effect. Traditional interpolation fails to produce a smooth result in this case (see Figure 8, bottom row).

6.3 Multiple objects

Finally, we demonstrate the ability of our system to constrain multiple objects in a scene. Figure 9 (top row) shows a scene with two tables side by side. The camera begins on one side of the tables, rotates as it passes overhead, and moves to the opposite side. The image-space trajectories of the two tables are adjusted so that they cross paths in the middle of the sequence as they both move from one side of the screen to the other.

It should be noted that when constraining multiple objects in a scene, it is possible for the animator to provide a set of constraints that are not satisfiable (see Figure 9, bottom row). In this example, the animator has moved the trajectories of the two objects in opposite directions at two control points, while leaving the size constraint the same. Our solver still returns a result that minimizes the total cost, but it does not precisely meet either the trajectory or size constraint.

7 Conclusion

We have presented a novel interface for using image-space constraints to control camera interpolation in animation sequences. Our interface makes it easy for the animator to specify the desired positions, orientations, and sizes of selected objects in image space, even if the objects are moving or changing size. We present an interactive, direct solver for constraining a single object, and a robust general solver for constraining multiple objects and manipulating internal camera parameters.

8 Acknowledgements

[Removed for blind review]

References

ALEXA, M. 2002. Linear combination of transformations. *ACM Transactions on Graphics 21*, 3 (July), 380–387.

ALICE, 2010. Website, http://www.alice.org/.

BARR, A. H., CURRIN, B., GABRIEL, S., AND HUGHES, J. F. 1992. Smooth interpolation of orientations with angular velocity constraints using quaternions. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, 313–320.

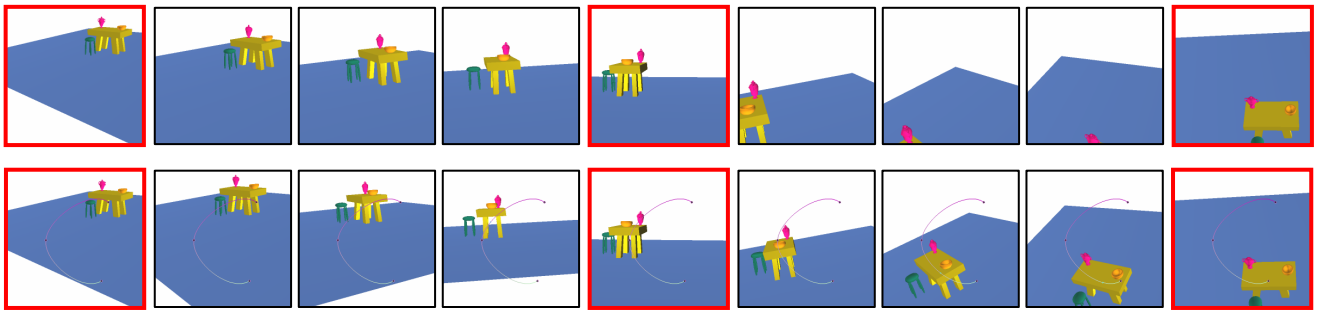

Figure 7: *A sequence which has no fixed focus point (3 keyframes – highlighted in red).* ***Top row:*** *Traditional interpolation interpolates the* ***camera's*** *3D position and orientation to produce in-between frames. Since the table is not located at the focus point, the interpolation results in the table rotating out of the viewing frame in the latter half of the sequence.* ***Bottom row:*** *We use image-space interpolation to interpolate the* ***table's*** *2D position, size, and orientation in the image. It would take the addition of several more keyframes to create a similar result using traditional interpolation.*

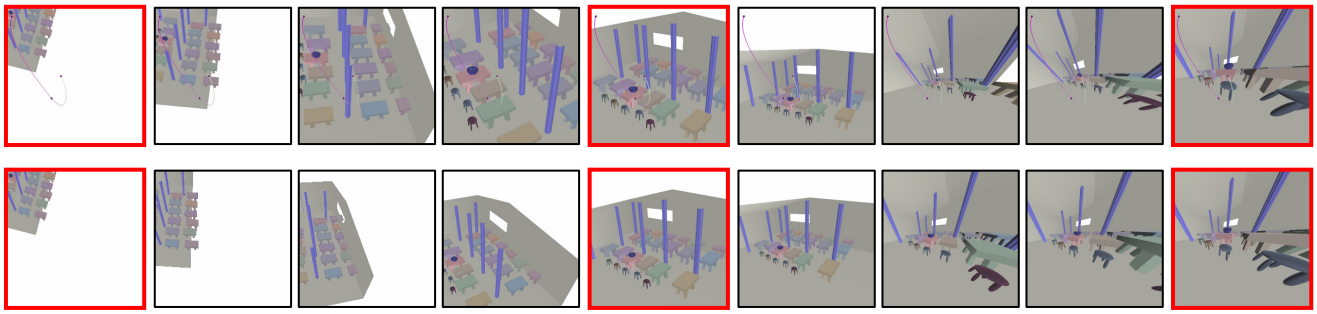

Figure 8: *Specifying a camera motion using three keyframes. The table rotates and gets larger between the first two keyframes. Both the distance to the camera and the zoom are changed between the second and third keyframes.* ***Top row:*** *The constraints are automatically generated from the keyframes. We used the general solver with the 12 DOF camera model and 120 frames.* ***Bottom row:*** *Traditional interpolation causes the table to rotate out of view and results in a zoom that is not visually smooth (see video).*

BARRETT, L., AND GRIMM, C. 2006. Smooth key-framing using the image plane. Tech. Rep. 28, Washington university in St. Louis.

BLINN, J. 1988. Where am i? what am i looking at? In *IEEE Computer Graphics and Applications*, vol. 22, 179–188.

DRUCKER, S. M., AND ZELTZER, D. 1995. Camdroid: A system for implementing intelligent camera control. In *1995 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 139–144. ISBN 0-89791-736-7.

GLEICHER, M., AND WITKIN, A. 1992. Through-the-lens camera control. In *Siggraph*, E. E. Catmull, Ed., vol. 26, 331–340. ISBN 0-201-51585-7. Held in Chicago, Illinois.

HARTLEY, R., AND ZISSERMAN, A. 2000. *Multiple view geometry*. Cambridge University press.

HARTLEY, R. I. 1992. Estimation of relative camera positions for uncalibrated cameras. In *ECCV '92: Proceedings of the Second European Conference on Computer Vision*, Springer-Verlag, London, UK, 579–587.

HAWKINS, A., AND GRIMM, C. 2007. Keyframing using linear interpolation of matrices. *Journal of Graphics Tools To appear*. Using linear matrix interpolation to do camera keyframing.

HOFER, M., AND POTTMANN, H. 2004. Energy-minimizing splines in manifolds. *ACM Transactions on Graphics 23*, 3 (Aug.), 284–293.

JACOBSON, D. H., AND MAYNE, D. Q. 1970. *Differential Dynamic Programming*. Elsevier.

KIM, M.-J., KIM, M.-S., AND SHIN, S. Y. 1995. A c2-continuous b-spline quaternion curve interpolating a given sequence of solid orientations. In *Computer Animation '95*.

LIAO, L.-Z., AND SHOEMAKER, C. A. 1992. Advantages of differential dynamic programming over newton's method for discrete-time optimal control problems. Tech. rep., Cornell Theory Center.

MICHENER, J. C., AND CARLBOM, I. B. 1980. Natural and efficient viewing parameters. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, vol. 14, 238–245.

NISTER, D. 2003. An efficient solution to the five-point relative pose problem. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '03)*, vol. 2, 195.

P. ABBEEL, A. COATES, M. Q., AND NG, A. Y. 2007. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems*, no. 19.

REID, G., TANG, J., AND ZHI, L. 2003. A complete symbolic-numeric linear method for camera pose determination. In *ISSAC '03: Proceedings of the 2003 international symposium on Symbolic and algebraic computation*, ACM Press, New York, NY, USA, 215–223.

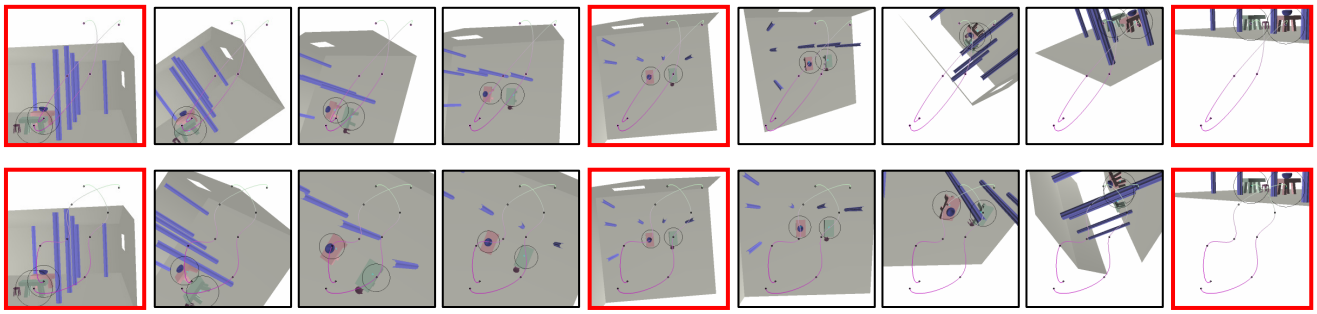

Figure 9: *Constraining multiple objects (3 positional keyframes – highlighted in red, 120 total frames, 7 DOF camera model).* ***Top row:*** *The two tables are made to cross paths as they move from one side of the screen to the other while the camera rotates overhead.* ***Bottom row:*** *The trajectories from the sequence in the top row have been pulled in opposite directions while the size constraint remains the same, yielding a set of constraints that are not satisfiable. The result is a sequence that minimizes total cost, but does not precisely meet the constraints.*

SHOEMAKE, K. 1985. Animating rotation with quaternion curves. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, vol. 19, 245–254.

SINGH, K., GRIMM, C., AND SUDARSANAM, N. 2004. The ibar: A perspective-based camera widget. In *UIST*. The first draft of the IBar.

SONG, D. M. 1993. Conics-based stereo, motion estimation, and pose determination. *International Journal of Computer Vision 10*, 1, 7–25.

SUTTON, R., AND BARTO, A. 1998. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

TASSA, Y., EREZ, T., AND SMART, W. D. 2008. Receding horizon differential dynamic programming. In *Advances in Neural Information Processing Systems 20*.

TOMLINSON, B., BLUMBERG, B., AND NAIN, D. 2000. Expressive autonomous cinematography for interactive virtual environments. In *Proc. of the 4th International Conference on Autonomous Agents*, ACM Press, 317–324.

TRIGGS, B. 1999. Camera pose and calibration from 4 or 5 known 3d points. In *Proceedings of the 7th International Conference on Computer Vision, Corfu, Greece*, 278–284.

WANG, L., PLESS, R., AND GRIMM, C. 2003. A 3d pattern for pose estimation for object capture. In *Vision Interface*, 395–401.

WEI HE, L., COHEN, M. F., AND SALESIN, D. H. 1996. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 217–224.

ZHANG, Z. 1999. Flexible camera calibration by viewing a plane from unknown orientations. In *ICCV*, 666–673.