Supplemental Document: Generalizing Wave Gestures from Sparse Examples for Real-time Character Control

Helge Rhodin[1], James Tompkin[2], Kwang In Kim[3], Edilson de Aguiar[4],
Hanspeter Pfister[2], Hans-Peter Seidel[1], Christian Theobalt[1]
[1]MPI for Informatics, [2]Harvard Paulson SEAS, [3]Lancaster University, [4]Federal University of Espirito Santo

1 Overview

We detail the system implementation, animation synthesis (Sec. 2), and experimental setup (Sec. 3). We also report additional user study findings (Sec. 4) and present pseudo-code of our algorithm to aid implementation (Sec. 5).

2 Animation synthesis

2.1 Background to parametrized motion graphs

Virtual characters often have their motion defined by an animation database, as is typical in video games. We organize an animation database into a *parametrized motion graph*. Motion graphs structure animation databases: Nodes in the graph represent decision points, and edges represent feasible transitions between motions. Rose et al. [1998] and Heck and Gleicher [2007] extend this structure where a node represents a motion class (e.g., walking) which is parametrized by motion properties such as speed or style.

In the main paper, we provide a two-dimensional example slice of the parametrized motion class of our dog character (frequency and phase dimensions; Fig. 3 in the main paper). In addition, Figure 1 in this document shows the emotion-frequency slice of the parameter space. Together, these form a 3D space for interpolating within.

Given the motion graph, output animations are synthesized by interpolating example motions at points within each parametrized motion class (node) and between nodes (along an edge). Such parametrized spaces are traditionally created by manual positioning of examples [Igarashi et al. 2005], which may require artistic skill or knowledge of the animation pipeline to produce good results. Automatic extensions include parametrized transitions [Shin and Oh 2006], dense graphs where possible transitions are stored for each database frame [Lee et al. 2010], statistical motion models [Min and Chai 2012], and interpolation and transition for mesh characters [Casas et al. 2012]. Graph construction has also been automated [Kovar et al. 2002; Heck and Gleicher 2007]. Animations are commonly synthesized according to high-level task constraints such as motion goals, and foot-plant and end-effector positions [Wiley and Hahn 1997; Lee et al. 2010; Levine et al. 2012; Lockwood and Singh 2012], though we focus on real-time gestural motion control.

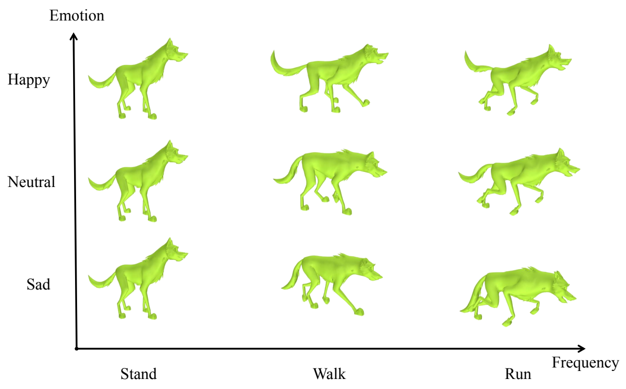

Figure 1: *Database character animations are structured into parameterized motion classes, such as this locomotion class for a dog. In the main paper we visualized the phase-frequency slice through the parametrized class, here we give the speed-emotion dimension.*

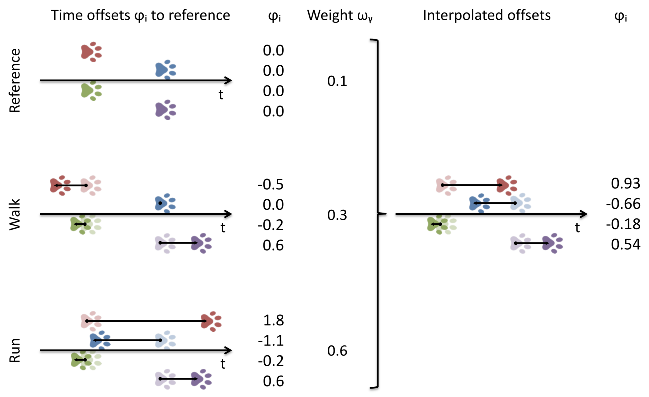

Figure 2: *Timeline example of the interpolation of three quadruped gaits with different beats. Footplant timing differences of the four limbs (red, blue, green, violet) are shown as shifts along the timeline. Temporal interpolation is executed on the time offsets φ_i to the reference gait with interpolation weights ω_Y for gait Y.*

2.2 Time interpolation

The main paper introduces how weights w_Y for each example animation Y are inferred and how phase values φ_i for each segment i are interpolated. The timeline in Figure 2 exemplifies this process with three quadruped gaits of different foot placement beats. Feet which are temporally aligned are unchanged; only those feet that have different beat patterns are treated. This time interpolation prevents strong artifacts in the subsequent mesh interpolation, such as foot sliding and hanging limbs (Fig. 3).

Quick changes in weights (i.e., in user control) can cause very abrupt changes in phase, even going back in time, e.g., when switching from a positive to a negative phase offset. To prevent this, we bound the phase change of individual segments to the global animation speed by enforcing $(1-0.05)(\varphi_t-\varphi_{t-1}) < (\varphi_{i,t}-\varphi_{i,t-1}) < (1+0.05)(\varphi_t-\varphi_{t-1})$, where subscript t denotes time and i is the segment index. This effectively prevents negative phase changes and bounds the abruptness of changes to the current motion speed, i.e., during slow motions only slow changes in relative time differences are permitted, while quick motions allow for quick adaption.

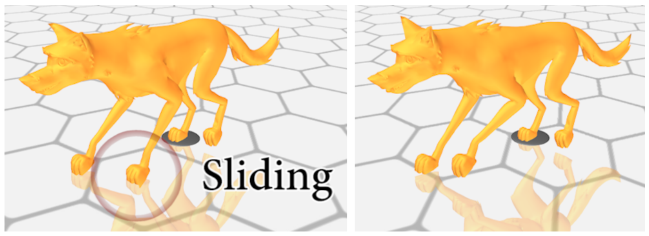

Figure 3: *Mesh interpolation with and without time interpolation of limb segments.* Left: *Without time interpolation, the front left leg shows sliding and hanging limb artifacts, as mesh interpolation requires roughly-aligned poses. This is violated by the front leg moving forwards in one animation to interpolate and backwards in the other in the global time-warp-aligned database animations.* Right: *The proposed separate alignment and time interpolation overcomes this limitation.*

2.3 Mesh pose interpolation

In this section, we provide details of the mesh interpolation given the previously-determined time offsets and interpolation weights w_Y. The interpolation of meshes is performed in a deformation space of per-triangle shear and rotation, with respect to a rest pose reference frame. A mesh segment A with F_A faces is represented by the set of shear matrices $\{S_{A,i} \in \mathbb{R}^{3\times3}\}_{i\in F_A}$ and rotation vectors $\{R_{A,i} \in \mathbb{R}\}_{i\in F_A}$ in axis-angle form.

Assume that two mesh segments A, B are interpolated based on weights w_A, w_B. The triangle transformations are linearly interpolated by $S_{C,i} = w_A S_{A,i} + w_B S_{B,i}$ and $R_{C,i} = w_A, R_{A,i} + w_B R_{B,i}$. From $S_{C,i}$ and $R_{C,i}$, a connected mesh is reconstructed from the differential coordinates of all segments by solving a Poisson system [Sumner and Popović 2004]. This established approach is a non-linear interpolation in vertex coordinates space, which runs in real-time for meshes of $10k$ triangles on a standard PC.

The individual mesh segments are masked by painting on the character's mesh (Fig. 4). We use masks with smooth boundaries and blend neighboring segments gradually to prevent seam artifacts. The interpolation in per-triangle rotation and shear space requires that rotations do not exceed $\pm180°$. In our experiments this was only violated at sparse points on the shoulder of the dog character, and originates from flipped triangles in the artist created database animation. For all other characters the example animations are without flipped triangles and their interpolation shows no seam artifacts.

2.4 Global translation interpolation

Consistency between the global motion of the animated character and the ground plane is important for realistic animation. To enable this, we annotate each animation in the database that contains global motion with the character velocity in the ground plane $[\mathbf{u}_0, \cdots, \mathbf{u}_T]$. We separate the character animation frames $[\mathbf{y}_0, \cdots, \mathbf{y}_T]$ from their global velocities $[\mathbf{u}_0, \cdots, \mathbf{u}_T]$ automatically by orthogonal Procrustes analysis [Sorkine 2009], which registers all frames against each other. During animation synthesis, we interpolate the velocities $u_{Y,\varphi}$ of all animation examples Y by the weights w_Y as before. The global position of the character is integrated from the interpolated velocities: it is the sum of all previous velocities multiplied by the respective changes in phase φ so as to accommodate for the current control frequency.

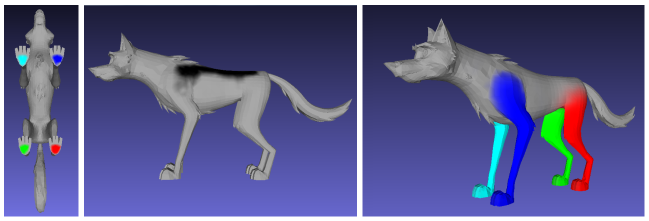

Figure 4: *From left to right; marked footplant constraint areas, torso spine region, and segmented limbs for the dog character.*

2.5 Foot plant constraints

The perceived realism of the synthesized animation also depends strongly on character feet not skating or sliding along the ground unexpectedly. To determine areas of the character that should be on the ground during the locomotion cycles (i.e., vertices at the bottom of each foot), these regions are manually marked in the character mesh. By painting in 3D, this takes about 1 minute per character. Further, a small area on the torso around the spine of the character location is marked, see Figure 4. This acts as an opposite constraint to maintain the character shape during mesh pose interpolation. Once completed, the interface propagates these vertices throughout each database sequence such that the ground contact can be annotated subsequently. This process additionally takes around one minute. For animations where the feet perfectly touch the ground during stance, the manual segmentation and annotation can be automated; however, this is usually not the case and the user must choose the desired planting behaviour when the feet are half-way down.

In the live phase, a character animation is generated by interpolating database animations in differential coordinates. This is performed in two passes: In the first pass, the character mesh is reconstructed from the interpolated differential coordinates, without any additional constraints. Then, the second pass adds target locations for the vertices which are marked in the preprocessing. The activation of the contact constraints is determined for each foot by the voting scheme as discussed in Section 7.3 in the main paper. In addition, we activate the foot constraints if the foot is close to the floor and has low velocity. This proximity heuristic triggers only in very few cases and is not reliable enough to replace the voting scheme.

All constraint vertices are positioned as follows. If the vertex constraint was activated at the previous frame, the vertex target is set to the previous location to pin the foot. If the constraint is newly activated at the current frame, the vertex target is set to the previous position with the vertical position set to the ground height to establish contact. The vertices of the feet which are in flight phase, i.e., are unconstrained, are set to their respective position from the first pass, unless they exhibit a large velocity due to a constraint in the previous frames. We clamp the velocity v of each vertex to $0.5v_{\text{first}} < v < 1.5v_{\text{first}}$, where v_{first} is the velocity of the respective vertex in the unconstrained animation from the first pass. This bounds the velocity to be similar to the unconstrained animation and effectively prevents temporal jitter of the legs in situations where the constrained foot strongly deviates from the unconstrained animation, e.g., due to rotation when running along an arc. To reduce the influence of the foot plant enforcement on the body pose, the spine segment is constrained to the first pass reconstruction at all times.

Each target position (for constrained vertices) is regarded as a hard constraint. In our implementation, we included this as soft-constraint into the Poisson solver, but with $10,000$ times higher weight than other constraints, which effectively renders it as a hard constraint.

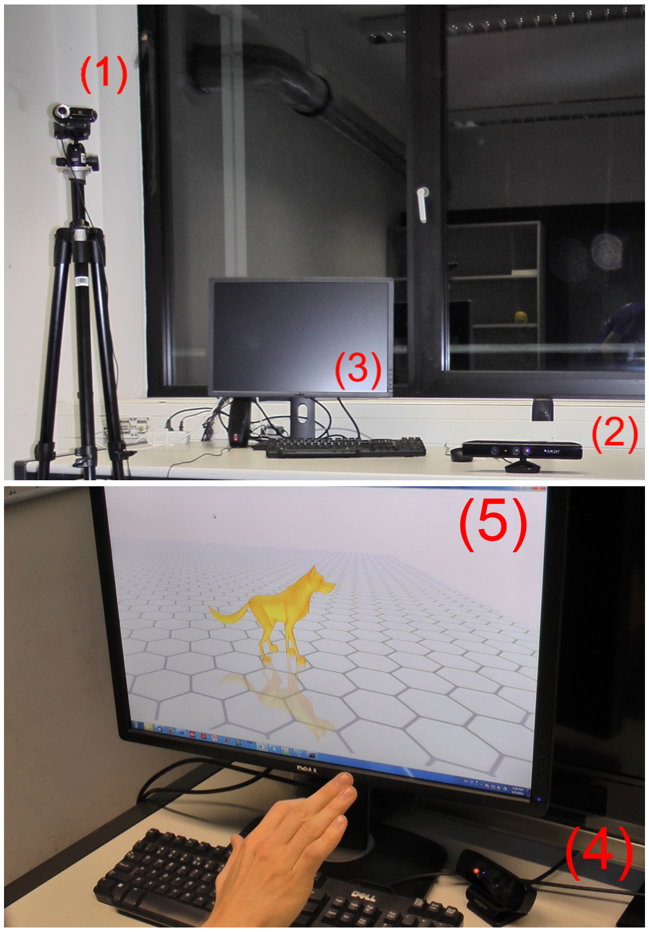

Figure 5: *Camera Setup: Facial expressions are recorded by a conventional webcam (1), full body motion by the Microsoft Kinect (2), hand motion by the Leap Motion sensor (4). The live animation is displayed on a LCD screen in front of the user (3,5).*

3 System setup

Our system is simple to setup as it requires only cheap low-quality motion capture sensors and is assembled within 5 minutes. In our experiments, full body motion is captured by the Microsoft Kinect sensor, hand articulation by the Leap Motion sensor, and facial motion by a conventional webcam with the Intraface software (Fig. 5). To control a new character the following steps are necessary:

Character animation. The controllable character motions are artist created and need to be acquired (we used 4-14 animations per character). Several commercial services offer game characters with basic motion cycles for free or low prices. As our system works on mesh representations any character that can be exported to a sequence of meshes can be used.

Database construction (takes 30-60 min.). Animations must be assigned to motion graph nodes, transition edges must be defined, and additive nodes must be marked (3 min.). Each animation needs to be annotated by semantic amplitude and frequency parameters (5 min., including refinement for novice users) and by foot-plant timings (1-3 min.). The foot-ground contact regions must be marked (2 min., once per character) and quadrupeds need to be segmented into torso and limb segments by painting (5 min., once per character).

Control definition (takes 1-15 min). Only a single reference control motion per motion graph node needs to be performed and recorded with the motion capture sensor. The recording of all control motions takes about one minute. Typically, the set of control motions is slightly refined to establish natural control for a specific character and to avoid ambiguous motions (15 minutes). In a game scenario the user could also select existing control motions out of a dictionary of predefined control motions (1 min).

The configuration of our characters is documented in Table 1 and database character animations are shown in a separate video.

4 User study tasks

The three game-like user study tasks are shown in Figure 7. In the following we give the exact task descriptions and explain the applied metrics. All results are reported in the main paper.

Task 1: Follow the shown path with the dog character without leaving it.

Quantitative metric 1:

- Number of times the path was left by the controlled character during completion of the course.
- Duration in seconds until the controlled character returns to the path after first leaving the path.

Task 2: Control caterpillar through obstacles (stones), start crawling, move through small gap, switch to walk after first stone gate, lift head fully after second stone gate, reduce head lift to half the height, and finally jump three times in a row.

Quantitative metrics 2:

- Number of trials until successful transition.
- Number of unintentional activated motions.
- Number of missed gates.
- Success of lifting head completely, then to half height.
- Difference from 3 jumps.

Task 3: Follow the leading horse, try to mimic its gait rhythm/beat and stride length as close as possible.

Quantitative metrics 3:

- Absolute difference between reference amplitude and user controlled amplitude
  (The reference was created by performance as well.)
- Absolute difference between reference gait frequency and user controlled frequency.

4.1 All results

In the main paper, we reported only the most relevant results of the user study. Figure 6 lists the complete results.

5 Pseudocode of the main components

To aid reproduction of the presented method, we provide pseudocode for the core components in Algorithms 1–5 in the following pages. We use the notation introduced in the main paper.

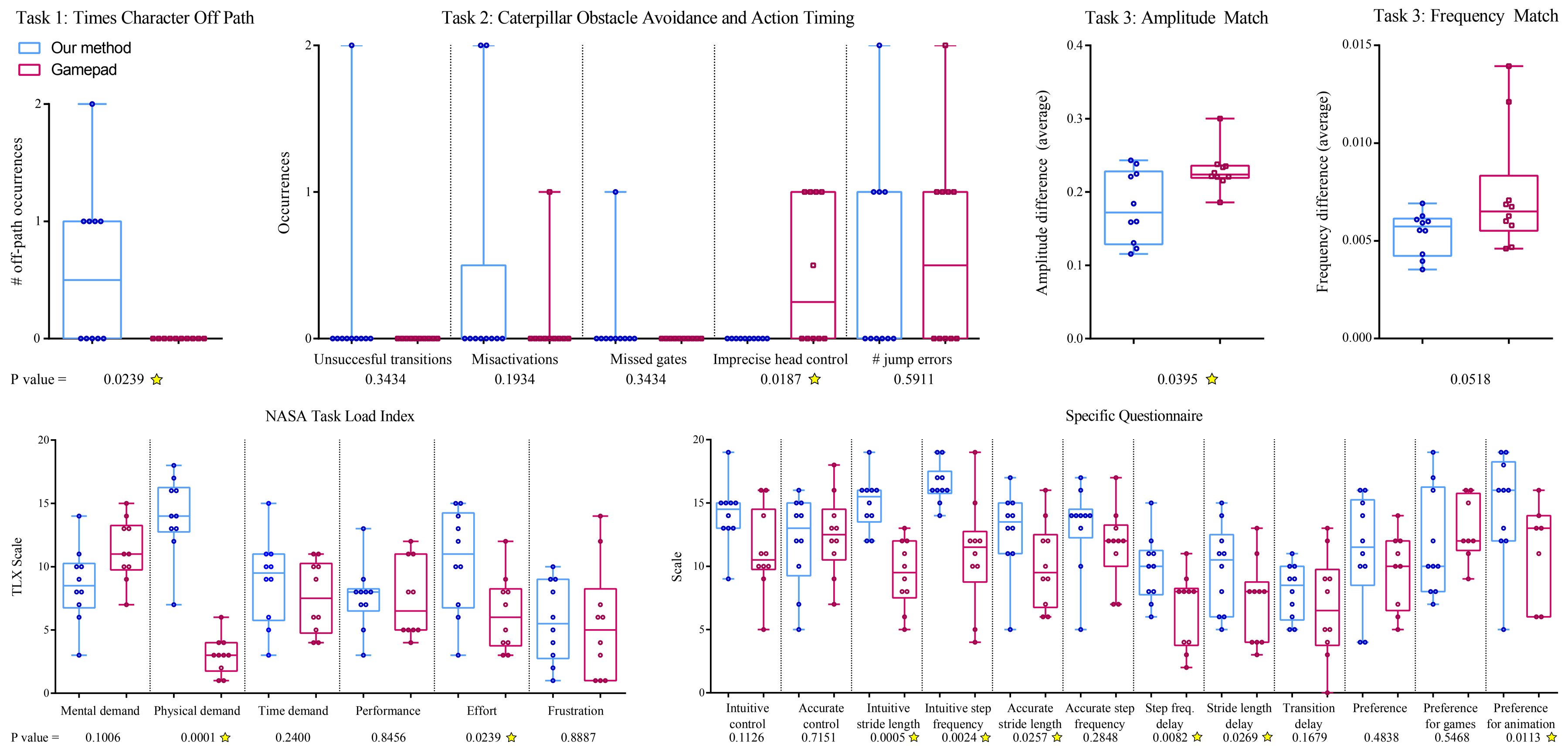

Figure 6: *Box and whisker plots for our task and questionnaire evaluation of our approach against a familiar gamepad controller. Two-tailed P values are provided from paired Student's t-test analysis, which significant at the 95% confidence level ($P < 0.05$) marked with a star.*

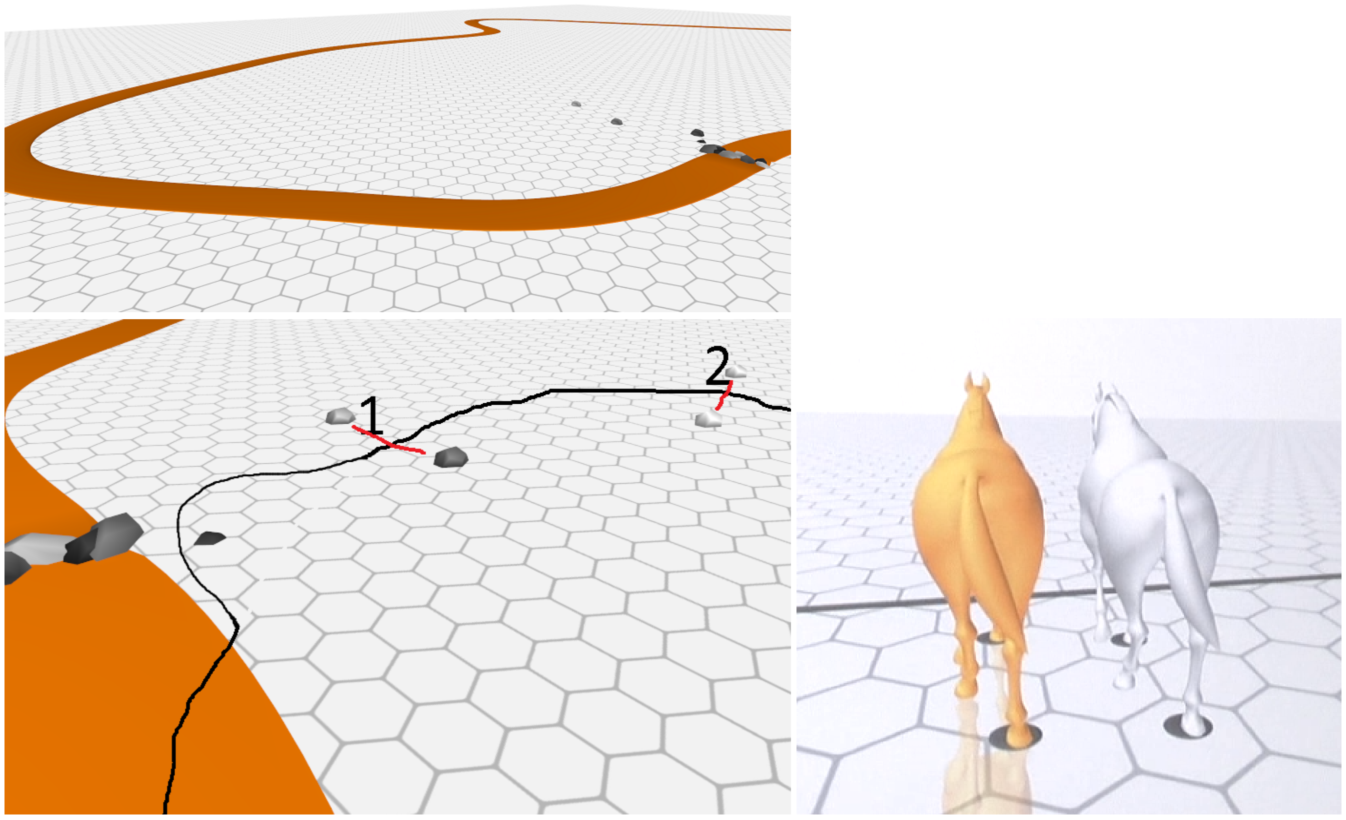

Figure 7: *User study tasks.* Top: *Task 1. Follow the path with the dog character, trying to stay within the path width.* Bottom left: *Task 2. Move through the terrain and gates, and switch motions with the caterpillar character when moving through each one.* Bottom right: *Task 3. Mimic the white horse by matching the step frequency, stride length, and dynamics (walk or trot or gallop).*

References

CASAS, D., TEJERA, M., GUILLEMAUT, J.-Y., AND HILTON, A. 2012. 4D parametric motion graphs for interactive animation. In *Proc. I3D*, 103–110.

HECK, R., AND GLEICHER, M. 2007. Parametric motion graphs. In *Proc. I3D*, 129–136.

IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. Spatial keyframing for performance-driven animation. *Proc. SCA*, 107–115.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. *ACM TOG (Proc. SIGGRAPH)*, 473–482.

LEE, Y., WAMPLER, K., BERNSTEIN, G., POPOVIĆ, J., AND POPOVIĆ, Z. 2010. Motion fields for interactive character locomotion. *ACM TOG (Proc. SIGGRAPH Asia) 29*, 6, 138:1–138:8.

LEVINE, S., WANG, J. M., HARAUX, A., POPOVIĆ, Z., AND KOLTUN, V. 2012. Continuous character control with low-dimensional embeddings. *ACM TOG (Proc. SIGGRAPH) 31*, 4, 1–10.

LOCKWOOD, N., AND SINGH, K. 2012. Finger walking: motion editing with contact-based hand performance. In *Proc. SCA*, 43–52.

MIN, J., AND CHAI, J. 2012. Motion graphs++: A compact generative model for semantic motion analysis and synthesis. *ACM TOG 31*, 6, 153–153.

ROSE, C., COHEN, M. F., AND BODENHEIMER, B. 1998. Verbs and adverbs: multidimensional motion interpolation. *IEEE CG&A 18*, 5, 32–40.

SHIN, H. J., AND OH, H. S. 2006. Fat graphs: Constructing an interactive character with continuous controls. In *Proc. SCA*, 291–298.

SORKINE, O. 2009. Least-squares rigid motion using SVD. *Technical notes, ETHZ*.

SUMNER, R. W., AND POPOVIĆ, J. 2004. Deformation transfer for triangle meshes. *ACM TOG (Proc. SIGGRAPH) 23*, 3, 399–405.

WILEY, D., AND HAHN, J. 1997. Interpolation synthesis of articulated figure motion. *IEEE CG&A 17*, 6, 39–45.

Table 1: *Specification of the annotation of the database animations for all tested characters. The motion class column specifies the motion graph node assignment, where each node is classified as cyclic or non-cyclic, and as primary or secondary.*

Character and database motion	Amplitude	Frequency	Emotion	Turn rate	Motion class	Is cyclic?	Is secondary?
Dog (no emotions)							
Stand	0.0	0.000	0	0	Locomotion	1	0
Walk neutral	0.7	0.020	0	0	Locomotion	1	0
Run neutral	0.8	0.040	0	0	Locomotion	1	0
Dog (with emotions)							
Stand	0.0	0.000	0	0	All classes	0,1	0
Walk happy	0.7	0.025	1	0	Locomotion	1	0
Walk neutral	0.7	0.025	0	0	Locomotion	1	0
Walk sad	0.7	0.025	-1	0	Locomotion	1	0
Run happy	0.7	0.050	1	0	Locomotion	1	0
Run neutral	0.7	0.050	0	0	Locomotion	1	0
Run sad	0.7	0.050	-1	0	Locomotion	1	0
Sit down	1.0	/	0	0	Sit	0	0
Shake slow	0.7	0.041	0	0	Shake body	1	1
Shake fast	0.7	0.083	0	0	Shake body	1	1
Look to side	1.0	/	0	0	Head motion	0	1
Wave paw	1.0	0.04	0	0	Wave paw	1	1
Scratch paw	1.0	/	0	0	Scratch paw	0	1
Talk	1.0	0.05	0	0	Talk	1	1
Horse							
Stand	0.0	0.000	0	0	Locomotion	1	0
Trot happy	0.5	0.025	1	0	Locomotion	1	0
Trot neutral	0.5	0.025	0	0	Locomotion	1	0
Trot sad	0.5	0.025	-1	0	Locomotion	1	0
Gallop happy	0.5	0.040	1	0	Locomotion	1	0
Gallop neutral	0.5	0.040	0	0	Locomotion	1	0
Gallop sad	0.5	0.040	-1	0	Locomotion	1	0
Caterpillar							
Rest	0.0	0.000	0	0	All classes	0,1	0
Crawl	0.5	0.025	0	0	Locomotion	1	0
Loop walk	0.5	0.020	0	0	Locomotion	1	0
Loop walk excited	0.8	0.020	0	0	Locomotion	1	0
Loop walk watch out	1.0	0.020	0	0	Locomotion	1	0
Rise head	1.0	/	0	0	Rise head	0	0
Jump	1.0	0.020	0	0	Jump	1	0
Bend body left	0.0	/	0	1	Bend body	0	1
Bend body right	0.0	/	0	-1	Bend body	0	1
Humanoid							
Stand	0.0	0.000	0	0	Locomotion	1	0
Walk slow	0.7	0.025	0	0	Locomotion	1	0
Walk normal	0.7	0.033	0	0	Locomotion	1	0
Walk fast	0.7	0.038	0	0	Locomotion	1	0
Run normal	0.7	0.420	0	0	Locomotion	1	0
Run fast	0.7	0.053	0	0	Locomotion	1	0
Dinosaur							
Stand	0.0	0.000	0	0	All classes	0,1	0
Walk	0.9	0.032	0	0	Locomotion	1	0
Jump	0.4	0.048	0	0	Jump	1	0
Bend body up	0.3	/	0	0	Bend	0	0
Tail left	0.3	/	0	-1	Tail	0	1
Tail right	0.3	/	0	1	Tail	0	1

Algorithm 1 In the live phase, **mapToIR** (frame by frame) maps the stream V of current user poses $\mathbf{x}_t$ to each of the intermediate representations (one per reference control motion X, Sec. 6.1 of the main paper). In the live step, the stream V of current user poses $\mathbf{x}_t$ is mapped to each of the intermediate representations (one per reference control motion X) by the method **mapToIR** (frame by frame). This step introduces independence of the control motions and reduces the dimensionality to simplify later filtering. It requires the current input poses $\mathbf{x}_t$, as well as the mapping functions Φ_X which are learned by ridge regression in the preprocessing step. The outputs are the individual intermediate representations and their estimated variances for the current time step.

```
function MAPTOIR($\mathbf{x}_t$)
    /* Loop over all reference control motions*/
    for all $X \in \mathcal{X}$ do
        /* Linear prediction */
        $\mathbf{z}_{X,t} = \Phi_X(\mathbf{x}_t)$

        /* Variance prediction (non-Gaussian error model) */
        nearestNeighbors = getNN($\mathbf{x}_t, X, 10$) /*Get the 10 nearest neighbors of $x_t$ in the reference control motion $X$ */

        /* Get corresponding reference intermediate representation (those used for training) */
        nearestReferenceIntermediateRep = getReferenceIntermediateRepresentations(nearestNeighbors)
        $\mathbf{zv}_{X,t}[1] = \frac{1}{|\text{nearestReferenceIntermediateRep}|^2} \sum_{\substack{\mathbf{z_a} \in \text{NearestNeighbors} \\ \mathbf{z_b} \in \text{NearestNeighbors}}} |z_a[1], z_b[1]|$ /* Average pairwise distance */
        $\mathbf{zv}_{X,t}[2] = \frac{1}{|\text{nearestReferenceIntermediateRep}|^2} \sum_{\substack{\mathbf{z_a} \in \text{NearestNeighbors} \\ \mathbf{z_b} \in \text{NearestNeighbors}}} |z_a[2], z_b[2]|$ /* Average pairwise distance */
    end for

    return $\mathbf{z}_{X_1,t}, \mathbf{z}_{X_2,t}, ..., \mathbf{zv}_{X_1,t}, \mathbf{zv}_{X_2,t}, ...$
end function
```

Algorithm 2 filterAmplitudeFrequencyAndPhase filters the intermediate representation. This improves the accuracy character control given noisy control motion (Sec 6.2 of the main paper). For robust and accurate character control, the intermediate representation is filtered by the method **filterAmplitudeFrequencyAndPhase**. It requires the stream of intermediate representations Z, represented as matrix with two rows and F=150 columns (frames), and their estimated variances ZV. The outputs are amplitude, phase, and frequency scalars.

```
function FILTERAMPLITUDEFREQUENCYANDPHASE(t,Z,ZV)
    $\tau = 150$
    $\mu = t$

    /* Loop over all frequency samples */
    for all $f \in [1/f_{t-1} - \tau, 1/f_{t-1} + \tau]$ do
        windowWidth $= \lambda/f$

        /* Compensate ambiguous estimates */
        var$_1$ = weightedAverageOverGaussWindow($ZV[1,\cdot]$, windowWidth)
        var$_2$ = weightedAverageOverGaussWindow($ZV[2,\cdot]$, windowWidth)
        $Z_{\text{normalized}}[1,\cdot] = Z[1,\cdot]2(\text{var}_1 + \text{var}_2)/\text{var}_1$
        $Z_{\text{normalized}}[2,\cdot] = Z[2,\cdot]2(\text{var}_1 + \text{var}_2)/\text{var}_2$

        /* Gabor filter */
        response[$f$] = $\langle [g(t-\tau;\mu,f),\ldots,g(t;\mu,f)],[z_{t-\tau},\ldots,z_t]\rangle$ /* Gives complex valued response */
    end for

    /* Return instantaneous frequency, amplitude, and phase */
    instantaneousFrequency = $\text{argmax}_f(|\text{response}[f]|_2)$
    instantaneousAmplitude = $\sqrt{|\text{response}[\text{instantaneousFrequency}]|_2} * \text{isSinusoidal}$
    windowEnergy = $\langle [N(t-\tau;\mu,\lambda/f),\ldots,N(t;\mu,\lambda/f)],[|z_{t-\tau}|,\ldots,|z_t|]\rangle$
    isSinusoidal = instantaneousAmplitude/windowEnergy
    instantaneousAmplitudeDamped = instantaneousAmplitude $* 2\max(0, \text{isSinusoidal} - 1/3)$
    instantaneousPhase = atan2(response[instantaneousFrequency, 2], response[instantaneousFrequency, 1])

    return instantaneousFrequency, instantaneousAmplitudeDamped, instantaneousPhase
end function
```

Algorithm 3 computeInterpolationWeights computes the interpolation weights in a single parametrized motion class (Sec. 7.2 of the main paper) The interpolation weights for interpolation inside a single parametrized motion class are computed by **computeInterpolationWeights**. The inputs to the method are the amplitude, frequency, phase, emotion, and heading direction parameters, as well as the predefined database annotations of the same quantities. The output is one scalar weight per database animation.

```
function COMPUTEINTERPOLATIONWEIGHTS($\theta, \theta_{Y_1}, \theta_{Y_2}, ...$)
    /* Loop over all database examples and compute weight by the inverse distance function */
    for all $Y \in \mathcal{Y}$ do
        $w_Y = \frac{1}{\|W(\theta-\theta_Y)\|_2} \left( \sum_Y \frac{1}{\|W(\theta-\theta_Y)\|_2} \right)$ /* Normalized weight */
    end for

    return $w_{Y_1}, w_{Y_2}, ...$
end function
```

Algorithm 4 interpolatePhase interpolates the phase data (the animation progression index). This step is performed before the mesh interpolation step (Sec. 7.2 of the main paper). Before pose interpolation, the phase, i.e., the animation progression index, is interpolated by **interpolatePhase**. The method inputs are the current global phase, its previous value, and the interpolation weights. The outputs are separate phase values per mesh segment (e.g., limbs).

```
function INTERPOLATEPHASE($\varphi_t, \varphi_{t-1}, w_{Y_1}, w_{Y_2}, ...$)
    /* Loop over all mesh segments */
    for all $i \in$ markedMeshSegments do
        $\varphi_{t,i} = \varphi_t + \sum_{Y \in \mathcal{Y}} w_{Y,t} \nabla \varphi_{Y,i}$

        /* Bound phase velocities by whole character velocity */
        $\varphi_{t,i} = \varphi_{t-1,i} + \max(0.95(\varphi_t - \varphi_{t-1}), \min(1.05(\varphi_t - \varphi_{t-1}), (\varphi_{t,i} - \varphi_{t-1,i})))$
    end for

    return $\varphi_{t,1}, \varphi_{t,2}, ...$
end function
```

Algorithm 5 interpolateMesh interpolates each mesh segment after phase interpolation (Sec. 2.3 of the supplemental document). Each mesh segment is interpolated individually by **interpolateMesh**. The input to the method are the separate phase values, as well as the interpolation weights. The output is the deformation gradient representation (per triangle rotation and shear matrices) of each mesh segment.

```
function INTERPOLATEPOSE($w_{Y_1}, w_{Y_2}, ..., \varphi_{t,1}, \varphi_{t,2}, ...$)
    /* Loop over database animations and interpolate triangle transformations*/
    for all $Y \in \mathcal{Y}$ do
        $S = \mathbf{0}$ /* List of shear matrices, all elements set to zero */
        $r = \mathbf{0}$ /* List of rotation vectors in axis angle form, all elements set to zero */

        /* Loop over all mesh segments */
        for all $i \in$ markedMeshSegments do
            /* Query deformation gradient rep. at time $\varphi_{t_i}$ from mesh segment i from database animation $Y$ */
            defGrad = $Y$.getMeshSegment($i$).getFrame($\varphi_{t,i}$)
            $S_i$ = defGrad.getShearMatrixList()
            $R_i$ = defGrad.getRotationList()

            /* Interpolate */
            $S = S + w_Y S_i$
            $r = r + w_Y R_i$
        end for
    end for

    return $S_1, S_2, ..., R_1, R_2, ...$
end function
```

Algorithm 6 reconstructConnectedMesh reconstructs a connected mesh by solving a Poisson system (Sec. 2.3 of the supplemental document).The input to the method is the deformation gradient representation for each mesh segment and footplant activations. The output is a single connected mesh represented by its vertex positions.

```
function RECONSTRUCTCONNECTEDMESH($S_1, S_2, ..., R_1, R_2, ...,$ footplantActivation)
    /* Concatenate segment representations */
    $S =$ concat$(S_1, S_2, ...)$
    $r =$ concat$(R_1, R_2, ...)$

    /* Construct differential coordinates from triangle transformations */
    differentialCoord = MeshLaplacian$(R(S($referenceMesh$))$

    /* Solve linear system to obtain unique vertex positions from differential coordinates */
    $\text{mesh}_{\text{unconstrained}} =$ solve(MeshLaplacian$x =$ differentialCoord$)$
    differentialCoord2 $=$ insertVertexConstraints(differentialCoord, $\text{mesh}_{\text{unconstrained}}$, footplantActivation$)$

    /* Solve linear system with augmented Laplacian to enforce footplant constraints */
    $\text{mesh}_{\text{constrained}} =$ solve(MeshLaplacian$_{\text{augmented}}$ $\text{mesh}_{\text{constrained}} =$ differentialCoord2$)$

    return $\text{mesh}_{\text{constrained}}$
end function
```