Machine Learning for Motion Synthesis and Character Control in Games

Michael Buttner, Unity Labs
Principal Research Engineer
michaelbu@unity3d.com
using UnityEngine;

namespace ThirdPerson
{

    [RequireComponent(typeof(Rigidbody))]
    [RequireComponent(typeof(CapsuleCollider))]
    [RequireComponent(typeof(Animator))]

    public class ThirdPersonCharacter : MonoBehaviour
    {

        void Start()
        {
            animator = GetComponent<Animator>();
            rigidbody = GetComponent<Rigidbody>();

            rigidbody.constraints = RigidbodyConstraints.FreezeRotationX | RigidbodyConstraints.FreezeRotationY | RigidbodyConstraints.FreezeRotationZ;

            originalGroundCheckDistance = groundCheckDistance;
        }

        private void FixedUpdate()
        {

            // Read current gamepad state
            //

            float h = Input.GetAxis("Left Analog Horizontal");
            float v = -Input.GetAxis("Left Analog Vertical");
        }

    }

}
Animation Graphs

Provide structure for animation clips

Allow animations to be “addressable”

Allow related animation to be grouped

Transitions have to be explicitly spelled out

Combinatorical explosion
Unstructured Animations

If we would combine all animation clips into a single unstructured library...

How can we infer information from the gameplay state directly?

Can we extract relevant information from the physical properties of the animations themselves?

How can we achieve the same level of versatility as animation graphs?
Motion Matching

Motion Matching [Michael Buttner, 2015]
Geometric Pose Comparison

- Minimum squared distance between joint positions
- Using only “relevant” joints
- Minimum local joint rotation delta is flawed
Motion Matching Algorithm

Motion Database

\[ \text{arg min}_x s(F(x), F(y)) \]

Query \( -> F(y) \)
Back-in-time Problem
Motion Matching

Pros
• Preserves high quality result
• Does not rely on phases
• Relatively easy to implement

Cons
• Prediction must match data
• Construction of cost functions
• Requires a lot of tweaking
• Doesn’t scale well
• Duplicate data problem
• “Back-in-time” problem
Motion Synthesis Research

Phase Function NN [Daniel Holden, 2017]

Deep Loco [Peng, 2017]

Mode Adaptive NN [Sebastian Starke, 2018]

Deep Mimic [Peng, 2018]

QuaterNet [Pavvlo, 2018]
The 4 No-No’s

- Locomotion & Cyclic motions
- Phase as temporal progression
- Pose merging
- Fitting animation along predicted trajectory
Locomotion & Cyclic motions

Most motion synthesis research emphasizes cyclic motions in general and locomotion in particular.
Phase as temporal progression

Most motion synthesis research uses the concept of a “phase”: scalar variable in the range $0$ to $2\pi$ representing the point in time of the current pose in the locomotion cycle.

Not true → Given a pose that corresponds to $\Theta$, all poses that correspond to $\Theta + \Delta t$ are “similar”.

“Phase” can have an arbitrary meaning (footcontact, entire “action” like for example cartwheel) – not a general concept.
Autoregressive methods

\[
X \begin{bmatrix}
0.8 \\
-0.3 \\
\vdots \\
0.65 \\
0.7
\end{bmatrix}
\]

\[
f
\]

\[
Y \begin{bmatrix}
0.15 \\
0.97 \\
\vdots \\
-0.03 \\
0.27
\end{bmatrix}
\]

\[
p_t
\]

\[
p_{t+1}
\]
Phase Function Neural Networks

A neural network where the weights are generated as a function of the phase

The “phase” is the scalar variable in the range 0 to $2\pi$ representing the point in time of the current pose in the locomotion cycle

$$W_i = \Theta_i(p)$$
Autoregressive methods & Pose merging

\[ f(x_0, x_{\ldots}, x_n, y_0, y_{\ldots}, y_m) \]

\[
\begin{bmatrix}
X \\
X_0 \\
\vdots \\
X_n \\
Y \\
Y_0 \\
\vdots \\
Y_m \\
\end{bmatrix} = \begin{bmatrix}
0.8 & -0.3 & \cdots & 0.65 & -0.7 \\
0.15 & 0.97 & \cdots & -0.03 & 0.27 \\
\end{bmatrix}
\]

\[ p_t \rightarrow f \rightarrow p_{t+1} \]
Autoregressive methods

Autoregressive network training averages the possible continuation candidates -> Loss of quality

“...NN is compact, requiring only a few megabytes of memory, even when trained on gigabytes of motion capture data...”

“...requires keeping all the motion data...”
Trajectory Control

Trajectory annotations are used to guide the pose generation process.

Expert gates can merging different movements (locomotion & jumping) -> Loss of quality.

Trajectory dictates timing, but instead animation needs to dictate overall timing.

![Graph showing trajectory annotations and timing](image-url)
Phase Function NN  |  Mode adaptive NN

Both approaches only work for locomotion
   Climbing for example has no obvious phase

Basic assumption – any pose that corresponds to
the same “phase” is similar

Poor quality
   “Floating”
   Don’t use exponential maps in NN’s

Variations won’t be preserved but get averaged

Memory footprint is determined by number of
weights – not the amount of animations used
   Neural Networks do not memorize anything!

Slow runtime performance
   my SSE implementation was ~0.9ms

4 samples of θ yields average result, more samples
require a higher memory footprint

Phase mispredictions result in a tendency
towards the mean pose

Extraordinary long training times (6+ hours)
   Edit data, retrain, hope it appears
   We can’t predict the output
   …or ask why it was produced

Mode adaptive NN’s don’t work for bipeds
The holy grail

- Fast turn-around times
- Ground-truth motion synthesis
- Minimal memory footprint
- Fast runtime
- Style

- Scaleable
- Controllable
- Versatile
- Precise
Motion Matching

Pros
• Preserves high quality result
• Does not rely on phases
• Relatively easy to implement

Cons
• Prediction must match data
• Construction of cost functions
• Requires a lot of tweaking
• Doesn’t scale well
• Duplicate data problem
• “Back-in-time” problem
A motion fragment is represented as a matrix $F_i \in \mathbb{M}^{\tau \times (m+1)}(\mathbb{R})$ where each entry $v_j^t$ contains the velocity of joint $j$ at some time $t \in \{i - \tau, \ldots, i + \tau\}$, and $j = 0$ represents the root transform.

$$F_i = \begin{bmatrix} v_0^i & v_0^{i+1} & \cdots & v_0^{i+\tau} \\ v_1^{i-\tau} & v_1^{i-\tau+1} & \cdots & v_1^i \\ v_2^{i-\tau} & v_2^{i-\tau+1} & \cdots & v_2^i \\ \vdots & \vdots & \ddots & \vdots \\ v_m^{i-\tau} & v_m^{i-\tau+1} & \cdots & v_m^i \end{bmatrix}$$

$$v_j^t = f_j^t - f_j^{t+1} \left( T_{r}^{t+1-1} T_{r}^{t} \right)$$
Kinematica Algorithm

\[ s(F, F') = \sum_{k=1}^{n} F_k - F'_k \]

\[ \text{arg min } x \ s(F(x), F(y)) \]

Query -> \( F(y) \)

Motion Database
Nearest Neighbor Search

$F_i = \begin{bmatrix}
    v_0^i & v_0^{i+1} & \cdots & v_0^{i+\tau} \\
    v_1^{i-\tau} & v_1^{i-\tau+1} & \cdots & v_1^i \\
    v_2^{i-\tau} & v_2^{i-\tau+1} & \cdots & v_2^i \\
    \vdots & \vdots & \ddots & \vdots \\
    v_m^{i-\tau} & v_m^{i-\tau+1} & \cdots & v_m^i 
\end{bmatrix}$

Product Quantization for Nearest Neighbor Search
[Jegou, Douze, Schmid, 2011]

$\hat{F}_i = \begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{bmatrix}$

300+ Scalar Values
1200 bytes per fragment
For 70,000 poses -> 82 Mb

< 64 bytes per fragment
For 70,000 poses -> ~3 Mb

Nearest Neighbor Search time < 0.05 ms (HP C#)
Sub-Linear Nearest Neighbor Search - $k = 1$
Short training time (< 5 minutes)
Abilities

Kinematica’s goal is to provide a **complete** alternative to animation graphs

Parkour, Climbing, Melee Combat, Synchronized movements, One-off actions, etc...

Prioritized list of abilities

Executed in order

This is not a super-imposed concept, i.e. Kinematica does not call into abilities
Kinematica Components

- Climbing
  - Mount
  - Drop down
  - Pull up
  - Free Climbing
  - Ledge Climbing

- Parkour

- Locomotion

Game Logic

Policies execute as part of the game code (ideally as C# jobs)

Policies execute variations of similarity searches depending on game logic

Similarity searches can be based on 1:1 or n:m fragments
Motion Library

All poses are arranged into a large matrix $D \in M^{(m+1) \times D} (\mathbb{R})$ where each column corresponds to a pose $J_i; i = 1, \ldots, m$

$$\begin{bmatrix}
T^1_r & T^2_r & \cdots & T^D_r \\
J^1_1 & J^2_1 & \cdots & J^D_1 \\
\vdots & \vdots & \ddots & \vdots \\
J^1_m & J^2_m & \cdots & J^D_m
\end{bmatrix}$$

Tagging segregates the motion library into addressable islands

Markers carry an arbitrary user-defined payload and are associated with discrete frames

Policies utilize tags and markers in user-defined similarity searches
Locomotion
Using our knowledge about the intended target location (NPC) or desired velocity (PC) we can generate a predicted path over the time horizon.
Collision detection

During the generation of the predicted future trajectory we can perform collision detection with the environment and other characters.
Trajectory Prediction

During trajectory prediction we use a character controller with the ability to forward simulate the collision world. This allows us to detect collisions in advance and plan accordingly.

Controller has full knowledge of which objects it collides with during normal frame-by-frame processing as well as during the prediction phase and can be safely rolled back in time to return to a previous simulation step.
Forward Prediction

- Snapshot
- Time Remaining > 0.0f
- Move(deltaTime)
- Rewind
- Yes
- No

Move(deltaTime) and Rewind are dependent on the time remaining. If the time remaining is greater than 0.0f, the system moves (deltaTime); otherwise, it rewinds.
Parkour
Anchors & Contacts

Parkour moves are designed to make precise contacts with the environment.

The goal is to generate a predicted future trajectory for a specific parkour move.

We use pose annotations to indicate which joint makes contact including the corresponding surface normal.

We denote the transform between the contact transform and the root transform of the first contact point as “anchor transform.”

Given contact transform $\rightarrow$ Move trajectory.
Anchors & Contacts

The goal is to generate a predicted future trajectory that “leads into” a specific parkour move.

In case the predicted future trajectory detects a collision during the prediction phase...

We generate a “contact transform” which in turn allows us to anchor the entire move in world space.

Now we can find possible transitions between the predicted future trajectory and the move trajectory.

We generate a new predicted future trajectory based on the result.
Climbing
Climbing is the most complex state

Several internal states

Transitions

Multiple movement types
Dismount
Drop down
Pull up
Mount
Free Climbing
Ledge Climbing
Dismount
Drop down
Mount
Free Climbing
Ledge Climbing
Pull up
Dismount
Drop down
Pull up
Mount
Free Climbing
Ledge Climbing
Climbing
Dismount
Drop down
Free Climbing
Pull up
Ledge Climbing
Mount
Ledge Climbing Model

Ledge geometry gets constructed on-the-fly

Ledge anchor is (line index, fraction)

Ledge anchor can be advanced based on distance

Ledge anchor can be constructed from world space position

Full predictive model

Snapshot() / Move() / Rewind()
Free Climbing Model

Wall geometry gets constructed on-the-fly

Wall anchor is (Normalized UV coordinate)

Wall anchor can be moved inside geometry bounds (2d displacement vector)

Wall anchor can be constructed from world space position

Full predictive model

Snapshot() / Move() / Rewind()
World Model

It is important to note that this is an unavoidable complexity for any non-trivial character navigation.

Any game will require this kind of structure in one form or another.
Machine Learning for Motion Synthesis and Character Control in Games

Michael Buttner, Unity Labs
michaelbu@unity3d.com