Rigging a Resident Evil

Inside the Bone Code of Operation Raccoon City

Ben Hanke

GDC 2012

Ben Hanke
@repstos

Software Engineer at Slant Six Games, 2007 – Present
- Worked on various engine features, tools and tech
- Resident Evil: Operation Raccoon City
- SOCOM: US Navy SEALs Confrontation

Immersive Education, 2001 – 2007
- Educational software using games technology
- Kar2ouche, MediaStage, MissionMaker

Oxford Brookes University, 1998 - 2001
- B.Sc (Hons) Intelligent Systems

Slant Six Games

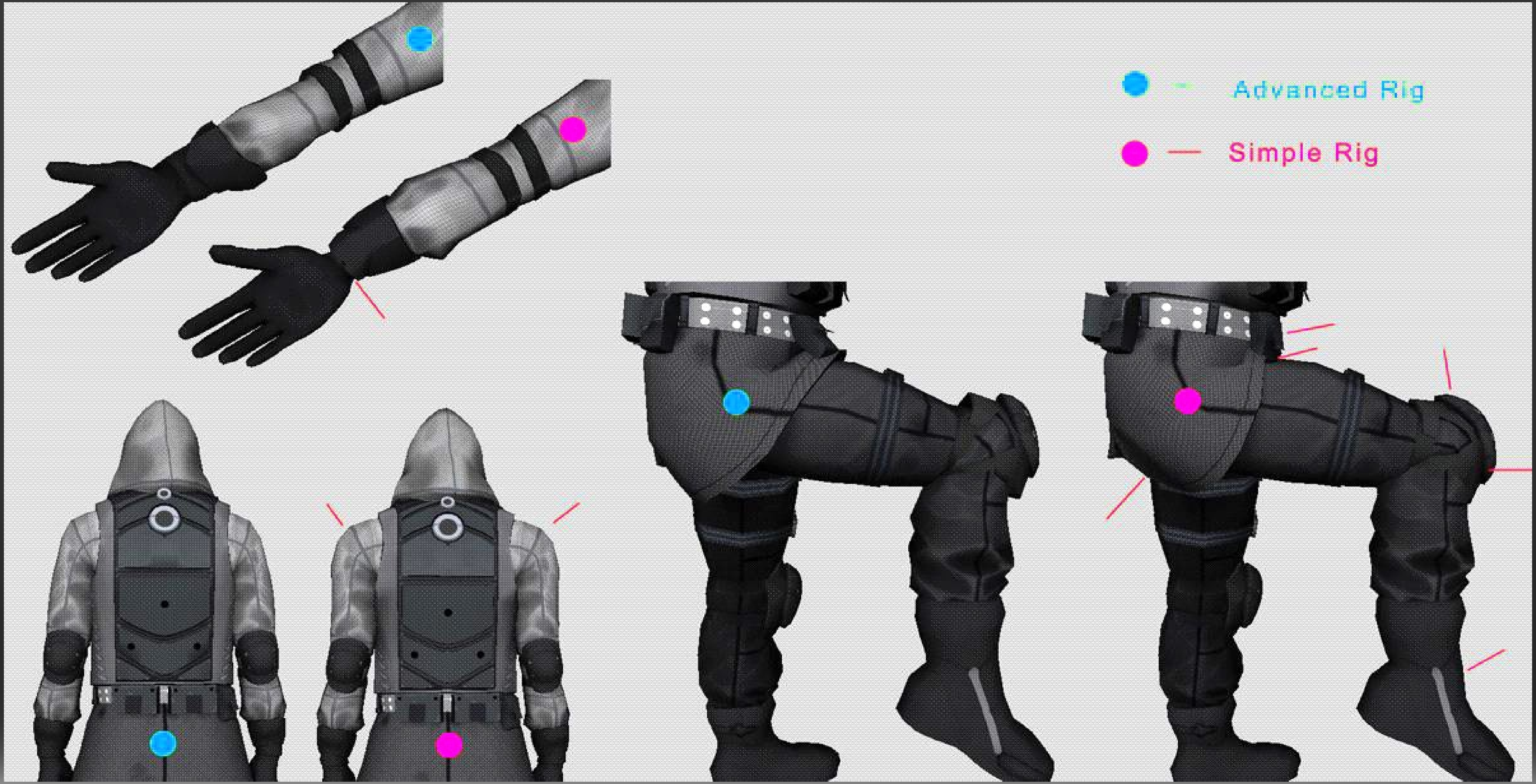
- Independent studio founded 2005
- Based in Vancouver, BC
- All Slant Six games are developed on our internal, multi-platform (PC, XBOX 360, PS3) engine technology
- Includes runtime, editors and toolsets for: Graphics, Animation, AI, Networking, UI, Core, High-level Gameplay

Overview

- Presenting a runtime solution for helper bones using Maya expressions
- Started as R&D project to improve character skinning for RE:ORC
- Saved us <u>lots</u> of time and memory in the long run
- Now a vital part of our animation engine
- Simple enough to explain in a lecture
- If you like it, you can do it too!

Requirements

Advanced vs. Simple Rig



Use Cases

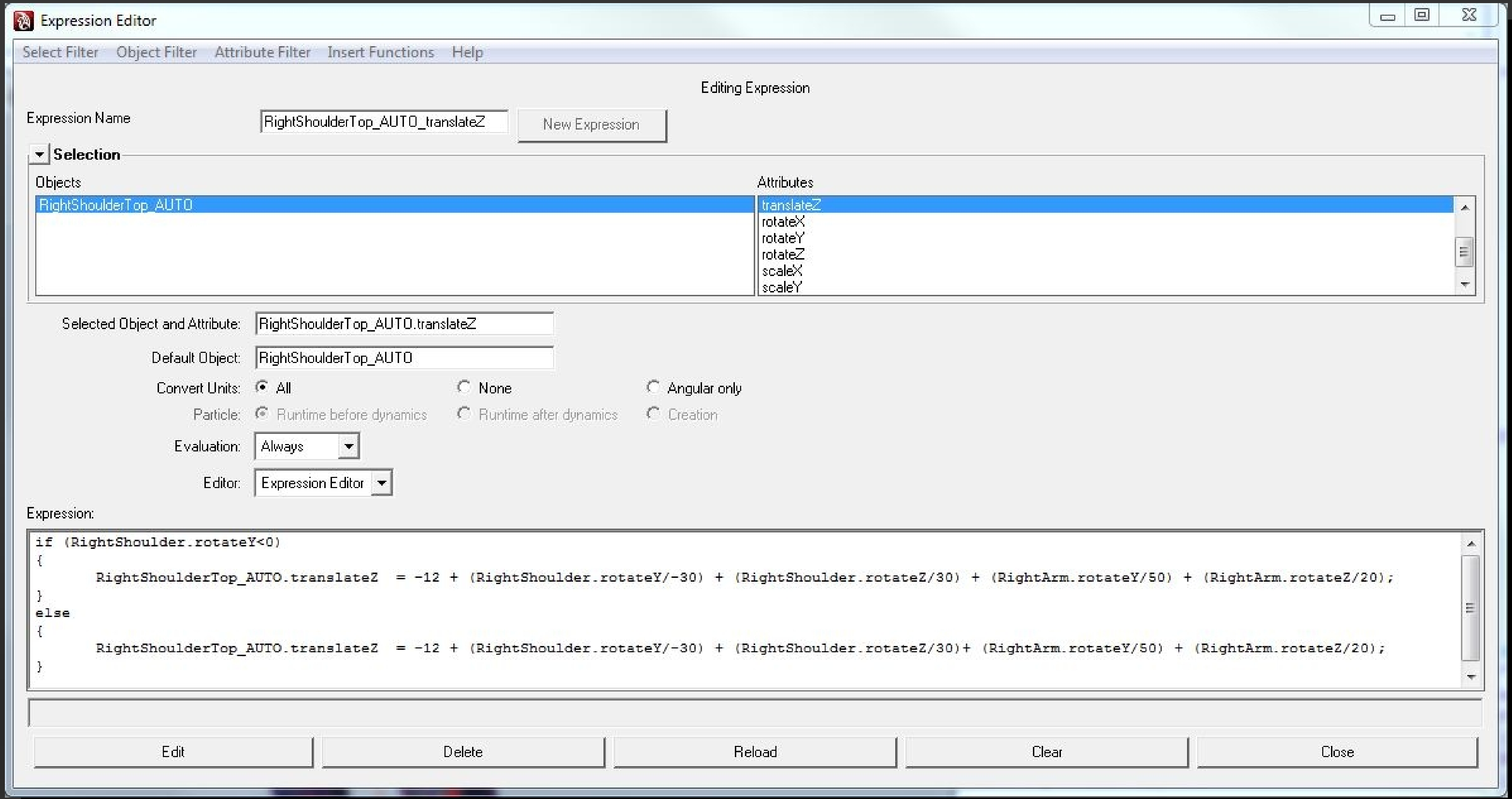
- One basic animation rig for animators.
- Additional, arbitrary helper bones for character artists.
- Overcome limitations of smooth blending across joints.
- Decouple animation and character rigging workflow.
- Clothing constraints: Skirts, collars, seams, sliding armour plates.
- Anatomic details: Twist bones for forearms, shoulder blades, biceps.
- Drive complex mesh from basic skeleton, e.g. hydraulic leg.

Maya Expressions

- Written in MEL (**M**aya **E**mbedded **L**anguage)
- Read/write local space joint transforms
- Variables: joint.attribute
  - `hips.translateY` = *[value in cm]*
  - `leftwrist.rotateX` = *[value in degrees]*
- One output, multiple inputs
- Can be interdependent

translate	X	Y	Z
rotate	X	Y	Z
scale	X	Y	Z

Authoring in Maya



[Authoring Demo]

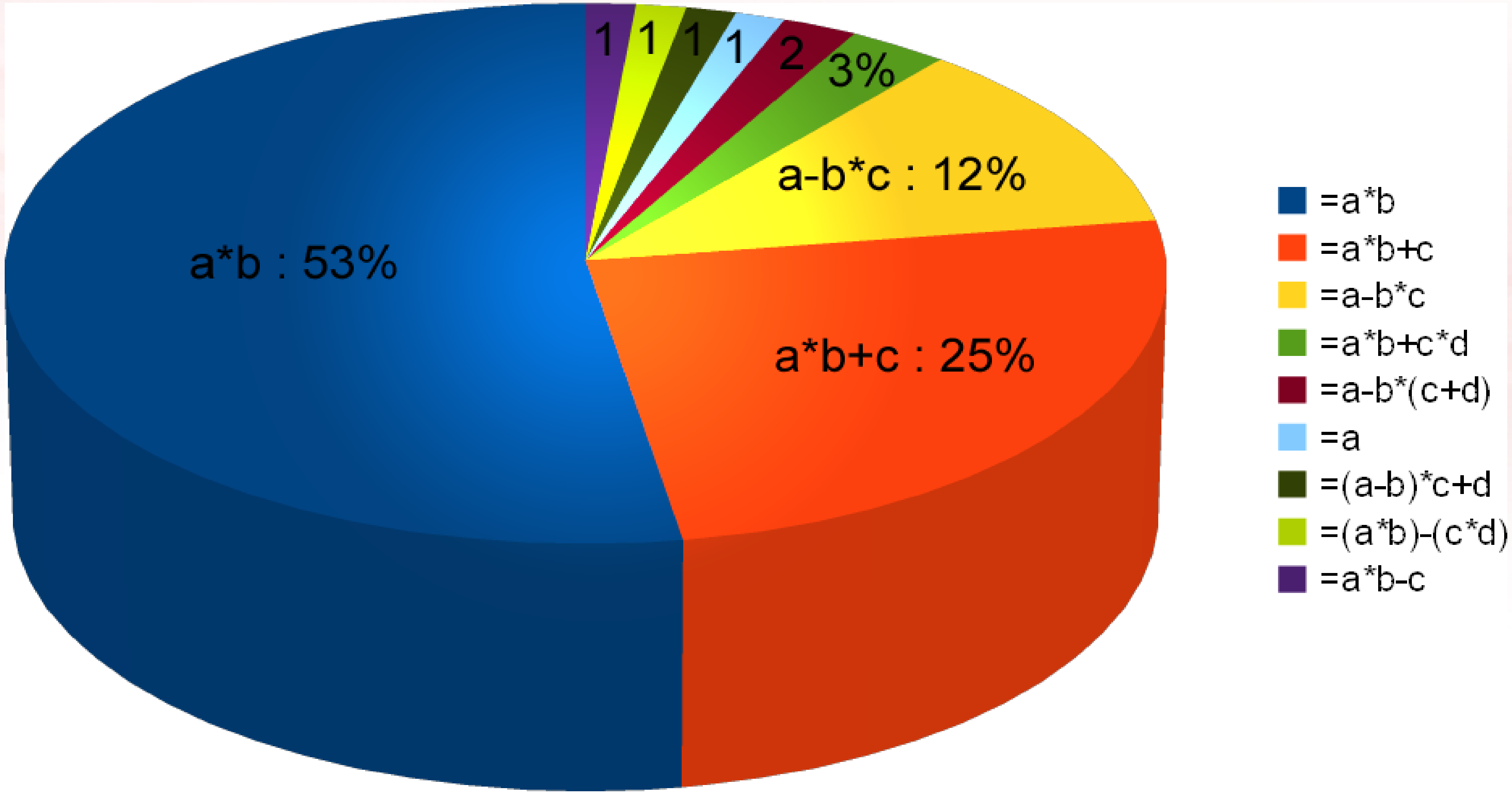
COLLADA Export

```
<expression
    id   = "RightBack_AUTO_rotateX"
    ixp  = ".O[0] = .I[0]/1.7;"
    o0   = "RightBack_AUTO.rotateX"
    i0   = "RightShoulder.rotateZ”
    init = "0"
/>
```

Test Data Analysis

- 117 expressions total, 20 branching.
- Canonicalized and sorted expression strings.
- Found lots of repetition.
- Most expressions very simple.
- Only one level of if/else branch
- Lots of division (slow and unsafe) :-(
- Division always by a constant :-)
- Refactored all division as multiplication.
- Found 9 function types (ignoring branches)

Function Frequency



Operations

Binary		
Add	add(a, b)	a + b
Subtract	sub(a, b)	a - b
Multiply	mul(a, b)	a * b
Divide	div(a, b)	a / b

Ternary		
Multiply & Add	madd(a, b, c)	a * b + c
Multiply & Subtract	msub(a, b, c)	a * b - c
Negative Multiply & Subtract	nmsub(a, b, c)	c – a * b

Test Data Analysis

- **91%** of expressions in Vector's rig achievable with just 1 instruction.
- Remaining 9% achievable with 2 instructions.

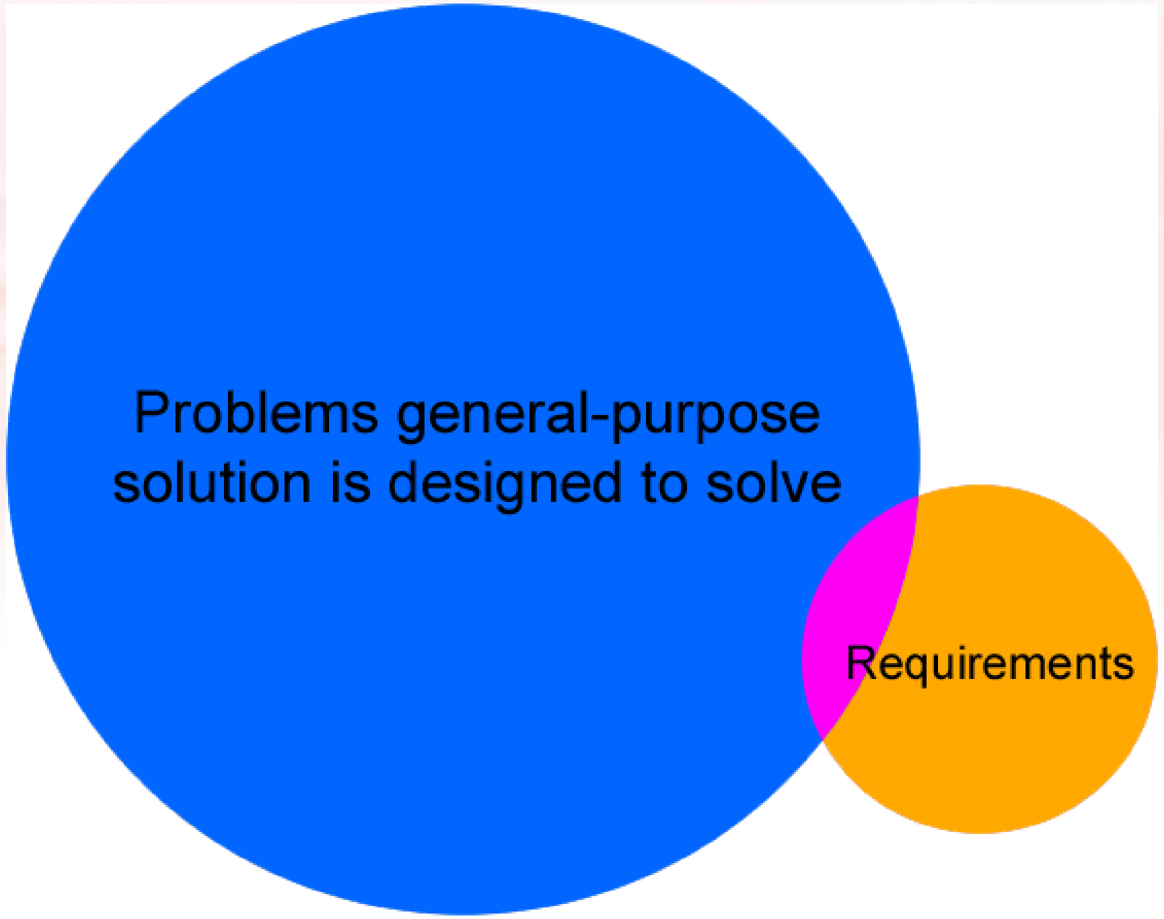
Expression Form	Functional Form
= a	= a
= a * b	= mul(a, b)
= a * b + c	= madd(a, b, c)
= a * b - c	= msub(a, b, c)
= c – a * b	= nmsub(a, b, c)
= a * b + (c * d)	= madd(a, b, mul(c, d))
= d – (a + b) * c	= nmsub(add(a, b), c, d)
= (a – b) * c + d	= madd(sub(a, b), c, d)
= a * b – c * d	= msub(a, b, mul(c, d))

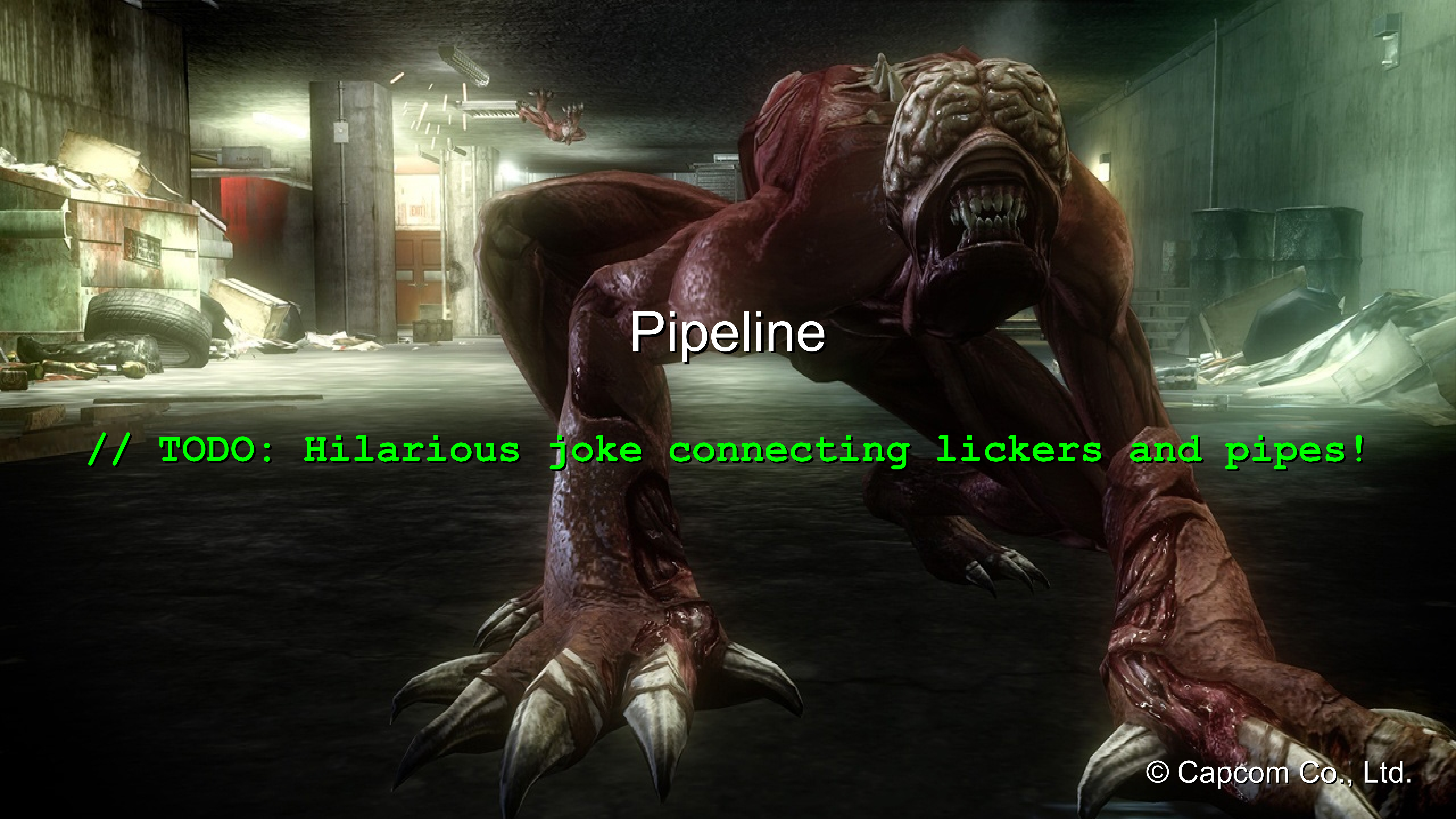
Branching

<table>
<tr><th>MEL Source</th><th>Functional Representation</th></tr>
<tr><td><pre>if(.i[0]>=25)
{
    .o[0]=((.i[0]-25)/4)+(.i[1]/-60);
}
else
{
    .o[0]=(.i[1]/-60);
}</pre></td><td><pre>select(
    cmpgte(.i[0], 25),
    madd(
        sub(.i[0], 25),
        0.25,
        mul(.i[1],
            -0.016666667)),
        mul(.i[1],
            -0.016666667)
   );</pre></td></tr>
</table>

Why Reinvent the Wheel?

- Considering existing solutions (Lua, Lex & Yacc, etc.)
  - What problems are they designed to solve?
  - How does that overlap with our requirements?
  - What are our constraints?
  - What new problems could they cause?





Pipeline

// TODO: Hilarious joke connecting lickers and pipes!

Parsing

- Command line program implemented in C#
- **<u>Not</u>** a general purpose compiler (cheat!)
- Tokenization limited to mathematical expressions – no MEL.
- High level syntax and patterns matched with RegEx.

```
szExpression        =    @"[a-z0-9.\(\)\[\]\+\-\*/]+";
szInputVariable     =    @"^.i\[[0-9]+\]";
szComparison        =    szExpression + "[<>=]+" + szExpression;
```

- **Pro Tip:** Use the **DebuggerDisplay** attribute!

Enums

```
public enum NodeType
{
    kExpression,         // one of the binary or ternary expressions defined in OperationType
    kConstant,           // e.g. "0.02"
    kVariable,           // e.g. ".i[0]"
    kCompareAndSelect,   // compares and selects based on one of the binary comparisons defined in ComparisonType
    kToken               // a temporary token node type used during parsing
};
```

```
public enum TokenType
{
    kOpenParentheses,              // "("
    kCloseParentheses,             // ")"
    kOperatorAdd,                  // "+"
    kOperatorSubtractOrNegate,     // "-"
    kOperatorMultiply,             // "*"
    kOperatorDivide                // "/"
};
```

```
public enum OperationType
{
    kAdd,      // a + b
    kSub,      // a - b
    kMul,      // a * b
    kDiv,      // a / b
    kMAdd,     // a * b + c
    kMSub,     // a * b - c
    kNMSub     // c - a * b
};
```

```
public enum ComparisonType
{
    kEqual,          // a == b
    kGreater,        // a > b
    kGreaterEqual,   // a >= b
    kLess,           // a < b
    kLessEqual       // a <= b
};
```

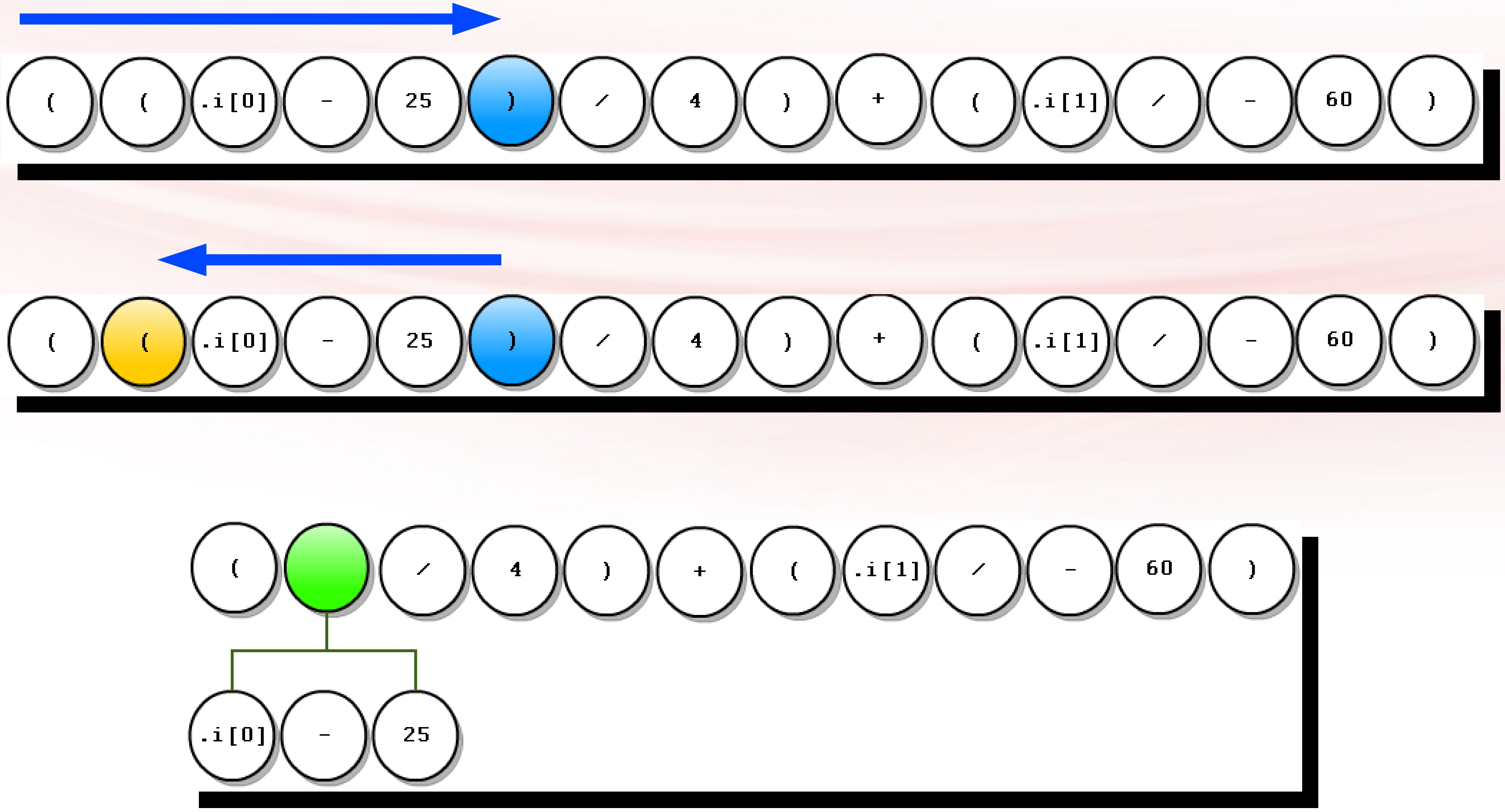
Tokenization

- Remove white space
- Detect and validate high-level control structure and output assignment
- Extract clean expression strings
- Classify and pop all tokens

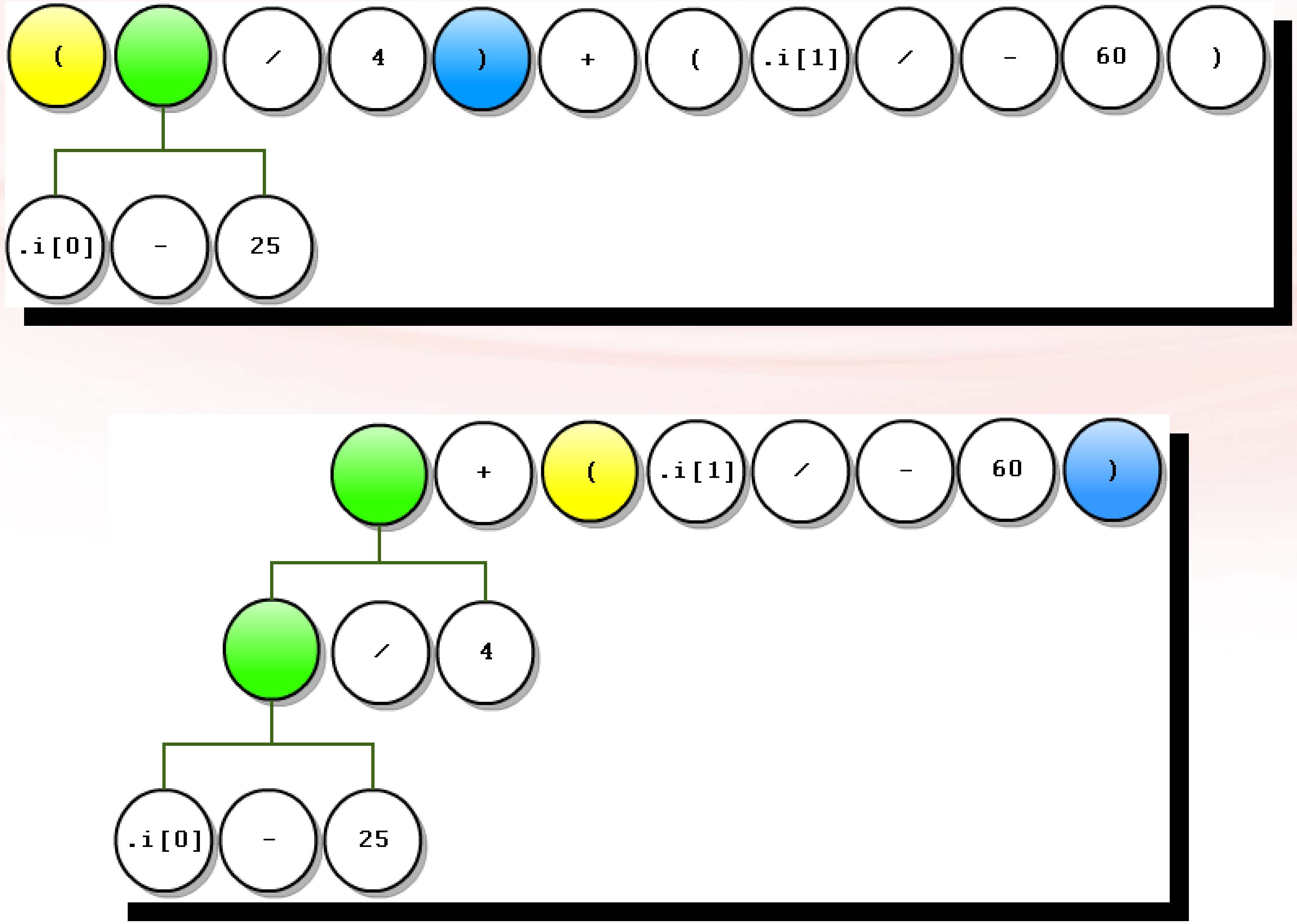
`@"^.o\[0\]"`	Throw exception (mustn't read output!)
`@"^.i\[[0-9]+\]"`	Pop **variable** node with index
`@"^[0-9]*\.?[0-9]+"`	Pop **constant** node with value
`(  ) + - * /`	Pop **token** node with type

- **Pro Tip:** Throw detailed exception messages for errors found during parsing.

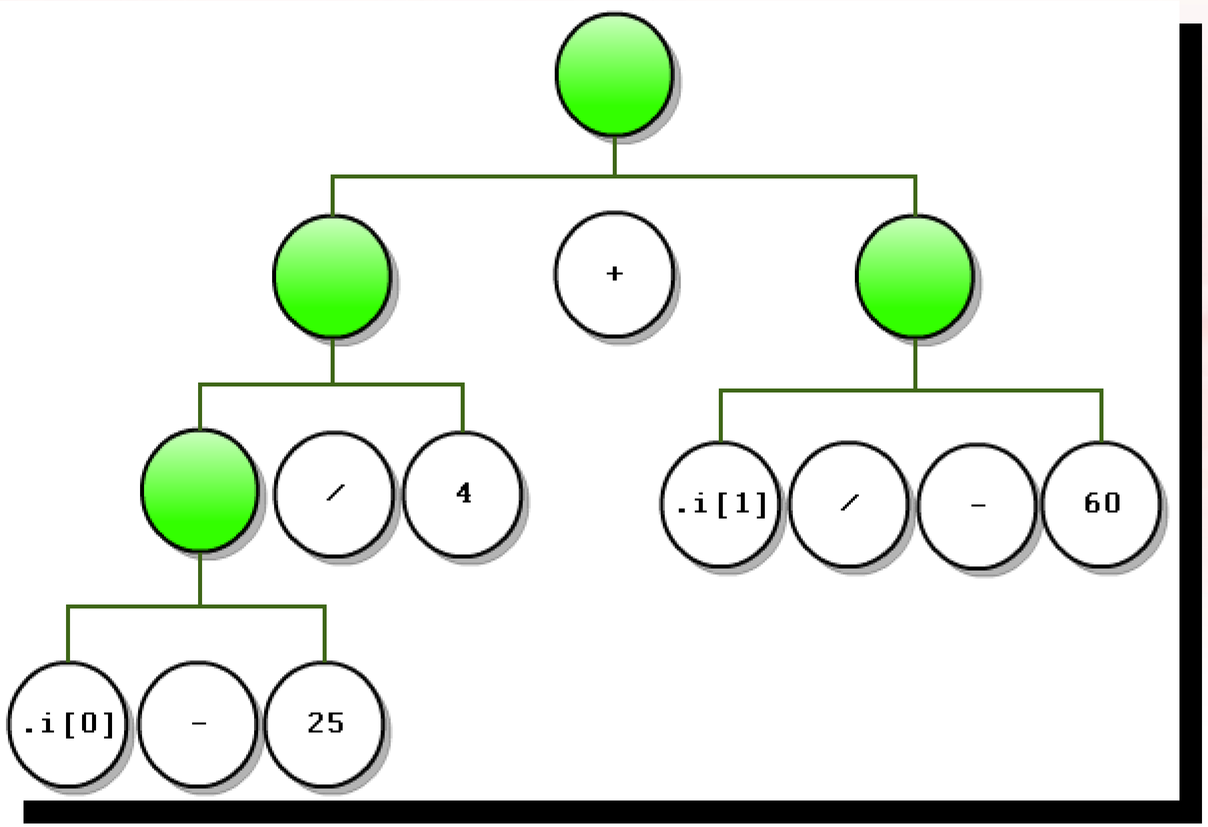
Collapse Parentheses



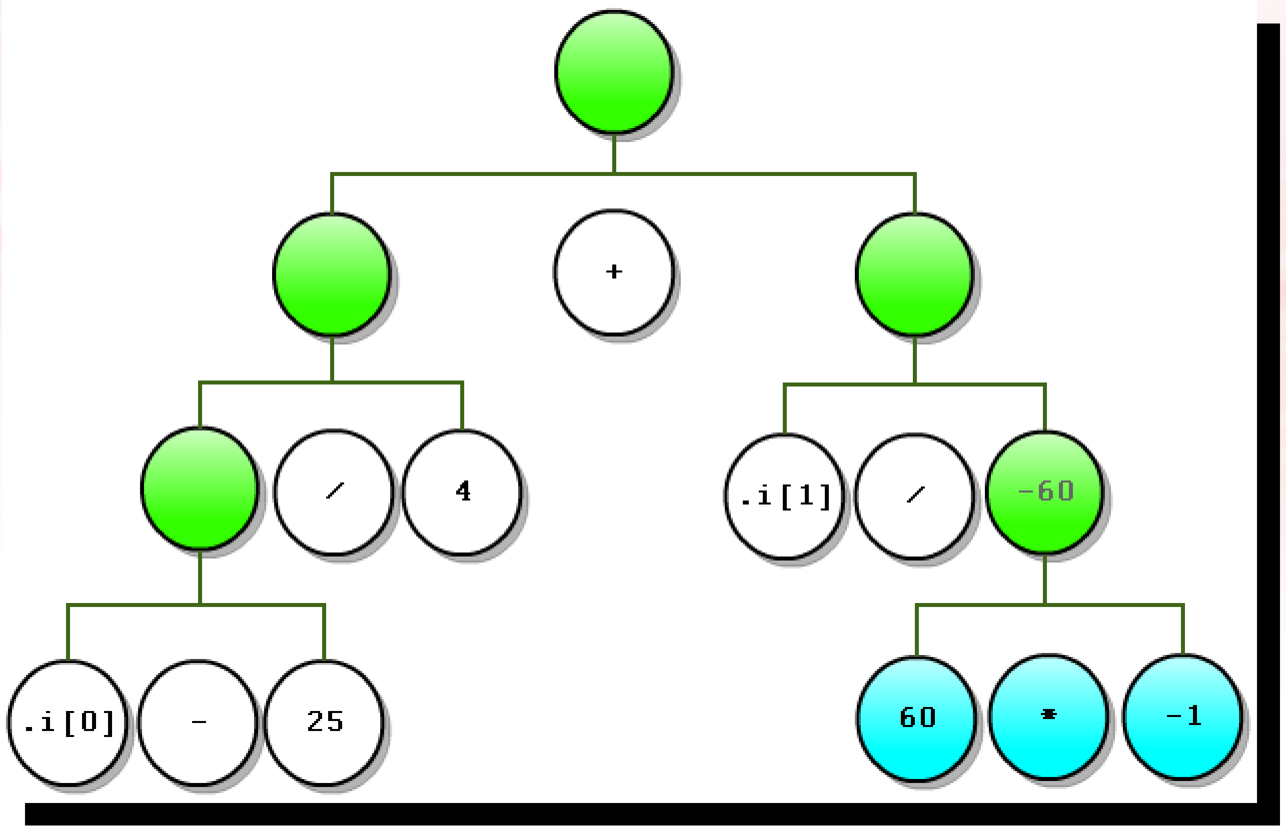
Collapse Parentheses



Collapse Parentheses

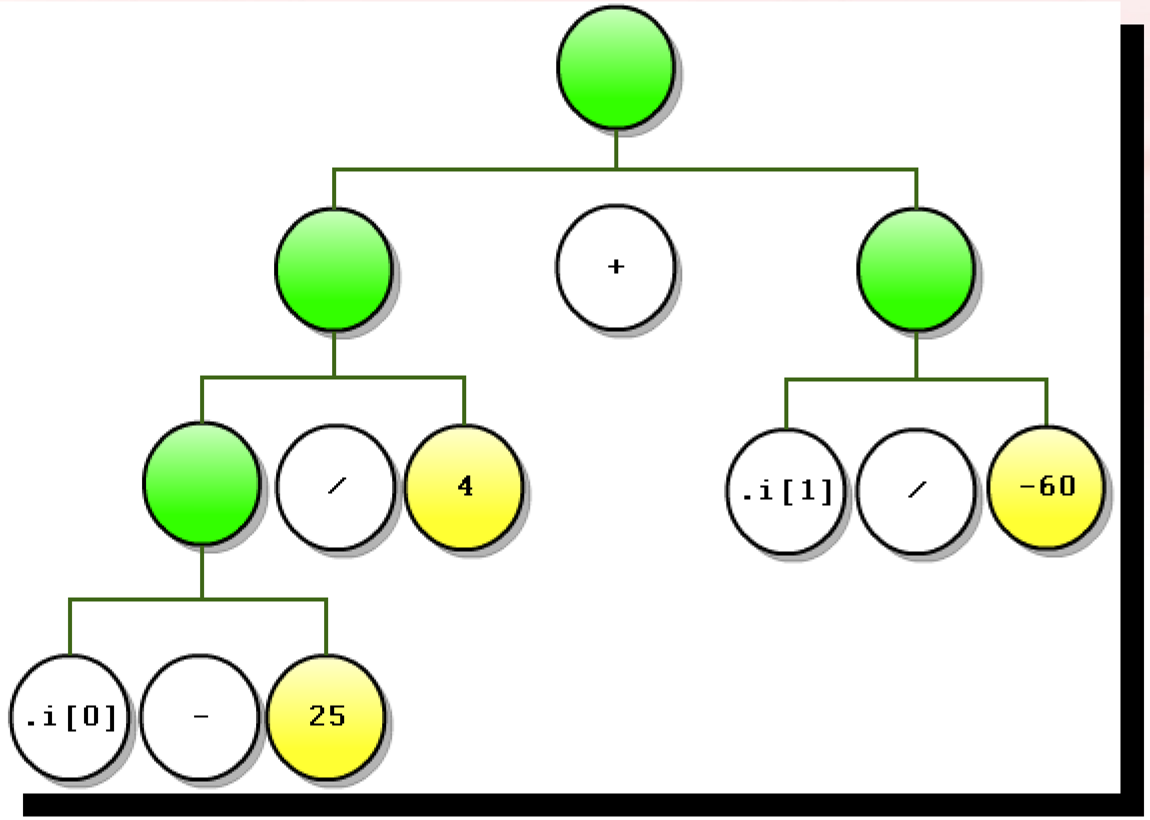


Refactor Unary Negation



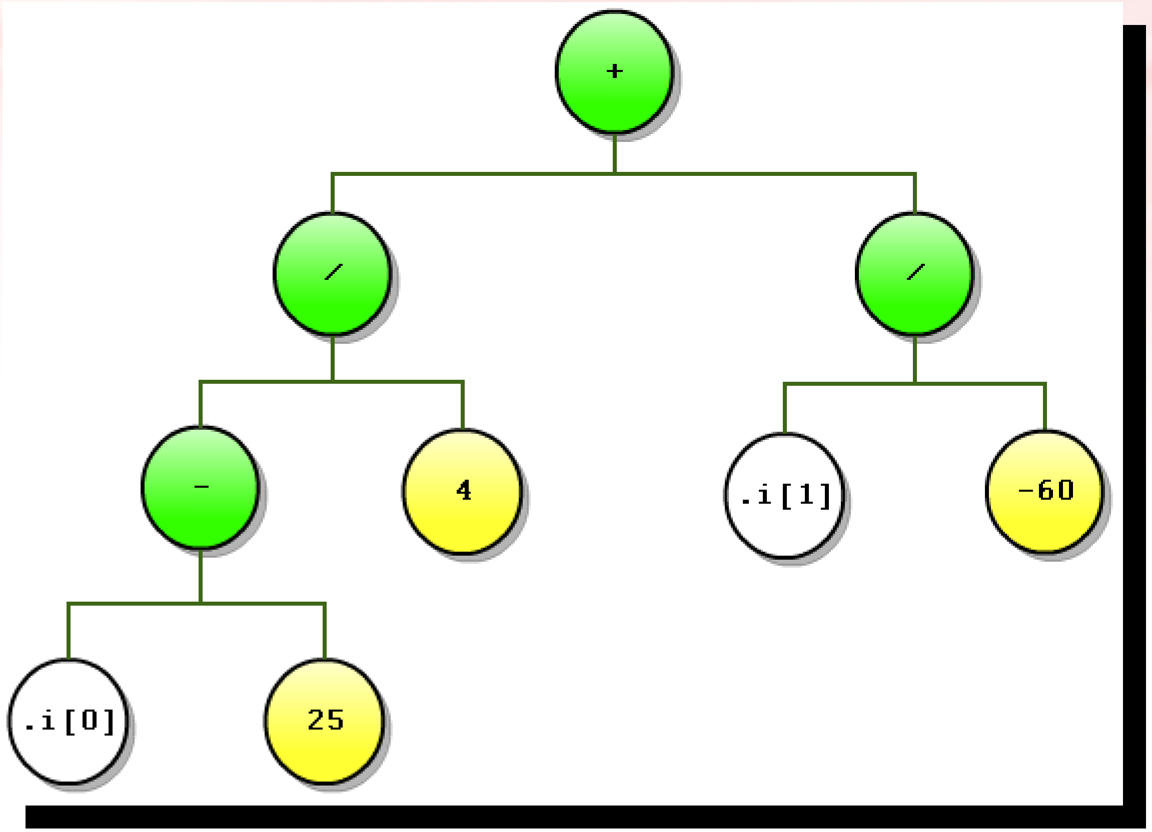
Bake Constant Expressions

- Recursively bake all constant expression nodes.
- Once this process is complete, all 'untyped' expression nodes contain at least one variable, one operator, and one other variable or constant.

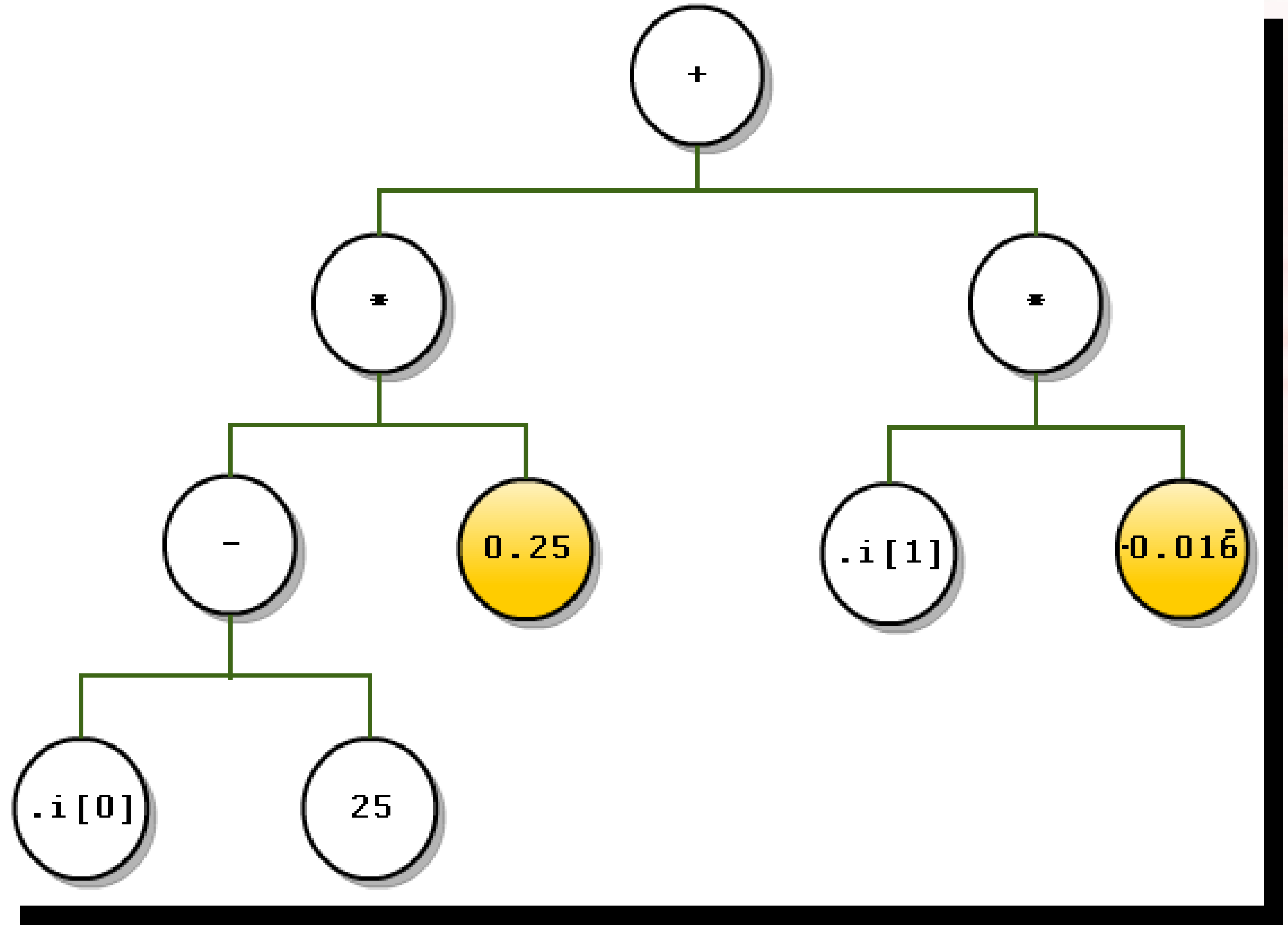


Binary Operators

- Convert all operator token nodes into binary expression nodes.
- Process * and / first, before + and -.
- All token type nodes now gone.



Refactor Division as Multiplication




```
Add(
    Mul(
        Sub( .i[0], 25.0f ),
        0.25f
    ),
    Mul( .i[1], -0.16666667f )
)
```

Collapse to Ternary

```
void CollapseToTernary_R()
{
    // Recurse depth first
    foreach (Node child in m_children)
        child.CollapseToTernary_R();

    if (m_nodeType == NodeType.kExpression)
    {
        switch (GetExpressionType())
        {
        case ExpressionType.kAdd:
            if (m_children[0].GetExpressionType() == ExpressionType.kMul)
                // add(mul(a, b), c) --> madd(a, b, c)
            else if (m_children[1].GetExpressionType() == ExpressionType.kMul)
                // add(c, mul(a, b)) --> madd(a, b, c)
          break;

        case ExpressionType.kSub:
            if (m_children[0].GetExpressionType() == ExpressionType.kMul)
                // sub(mul(a, b), c) --> msub(a, b, c)
            else if (m_children[1].GetExpressionType() == ExpressionType.kMul)
                // sub(c, mul(a, b)) --> nmsub(a, b, c)
          }
          break;
        }
    }
}
```

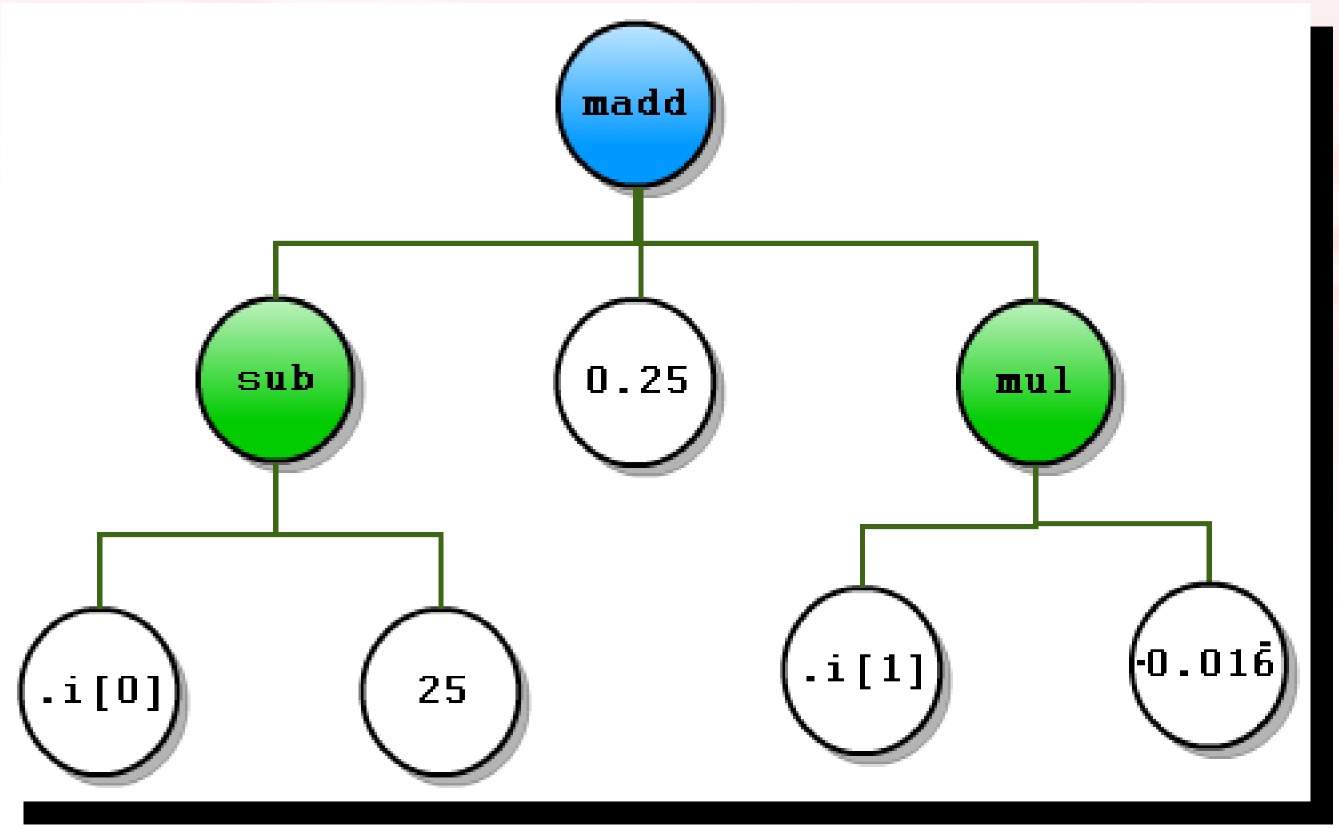
add(mul(a,b),c) → madd(a,b,c)

add(c,mul(a,b)) → madd(a,b,c)

sub(mul(a,b),c) → msub(a,b,c)

sub(c,mul(a,b)) → nmsub(a,b,c)

Final Expression Tree




```
((.i[0]-25)/4)+(.i[1]/-60)
```

```
madd(
    sub( .i[0], 25.0f ),
    0.25f,
    mul( .i[1], -0.016666667f )
)
```

Optimization

- Examine your input and output
- Bytecode length → Runtime cost
- Look for patterns that waste cycles
  - Constant expressions
  - Multiplication by 1 or 0
  - Identical if/else branches
- Trim nodes
- Add new fixed functions

Binary Export

- Sort expressions for writing according to dependencies.
- Count unique constants in expression and assign indices.
- Write unique joint hash name, index and constant arrays.
- Walk expression tree to write instructions in runtime evaluation order.
  - Breadth first for recursive.
  - Depth first for iterative.
- Each node is stored two values packed into a byte as (type | arg)

Node Type (2 bits)	**Argument (up to 6 bits)**	**Arity**
Expression	Operation type, e.g. Madd	2 or 3 (binary or ternary op)
Variable	Variable index	0
Constant	Constant index	0
CompareAndSelect	Comparison type, e.g. GreaterThan	4 (lhs, rhs, if branch, else branch)

Example Output

source	if (.I[0]<0){.O[0] = -12 + (.I[0]/-30) + (.I[1]/30) + (.I[2]/50) + (.I[3]/20);}else{.O[0] = -12 + (.I[0]/-30) + (.I[1]/30)+ (.I[2]/50) + (.I[3]/20);}
constants	0.05, 0.02, 0.0333333, -0.0333333, -12
output	RightShoulderTop_AUTO.translateZ
inputs	RightShoulder.rotateY, RightShoulder.rotateZ, RightArm.rotateY, RightArm.rotateZ
compiled	madd(.i[3], .c[0], madd(.i[2], .c[1], madd(.i[1], .c[2], madd(.i[0], .c[3], .c[4]))))
bytecode	4, 35, 16, 4, 34, 17, 4, 33, 18, 4, 32, 19, 20

source	if (.I[0]<0){.O[0] =(.I[0]/-5)+ (.I[1]/-3)+(.I[2]/-40);}else{.O[0] =(.I[0]/3)+(.I[1]/-3)+(.I[2]/-40);}
constants	0, -0.025, -0.2, -0.333333, 0.333333
output	RightShoulderTop_AUTO.rotateX
inputs	RightShoulder.rotateY, RightShoulder.rotateX, RightArm.rotateX
compiled	select(cmplt(.i[0], .c[0]), madd(.i[2], .c[1], madd(.i[0], .c[2], mul(.i[1], .c[3]))), madd(.i[2], .c[1], madd(.i[0], .c[4], mul(.i[1], .c[3]))))
bytecode	51, 32, 16, 4, 34, 17, 4, 32, 18, 2, 33, 19, 4, 34, 17, 4, 32, 20, 2, 33, 19

Runtime

Engine Integration

- Pose format must be convertible to and from Maya representation.
  - Model space matrices → local space Euler angles, model space translations.
- Suitable for asynchronous, parallel jobs.
  - e.g. SPURS on PS3, thread pools on X360/PC.
- Jobs small and self-contained, so easy to hide latency.
- Kick when animation pose is ready.
  - After pose blending, IK, ragdoll, NIS streaming, facial animation, etc.
- Results deadline: in time for skinning on GPU/SPU.

Job I/O

- Shared, read-only inputs (cacheable):
  - Expression data (Average ~10kB)
  - Bind pose and parent indices (Average ~10kB)
  - Anim-to-render skeleton index remapping table.
- Current animation pose [read-only] (68 bones for player character)
- Output render pose [write-only] (Average = 147, max = 176)
- Typically ~40kB local store required on SPU.
- Animation pose must be read-only after jobs are queued.
- Render pose must not be read until jobs are complete.

Converting Pose to 'Maya Space'

- Your Maya units may vary (we use degrees and centimeters).
- Convert translation to local space:

```
jointMat * Inverse( parentMat )
```

- Undo the bind pose rotation:

```
jointMat * Inverse( bindPoseMat * parentMat )
```

- Convert rotation to Euler angles in degrees: rotateX, rotateY, rotateZ
- Convert translation to centimeters: translateX, translateY, translateZ

- The reverse transform is simply:

```
rotation * bindPoseRotation * translation * parent
```

- Subtract the original bind pose value of the expression as exported from Maya!

Evaluating Expressions

- Simple recursive or iterative virtual machine where each iteration/call:
  - Consumes a byte
  - Unpacks node type and payload value, e.g. variable[index]
  - Switches on node type
  - Calculates result for parameter nodes as determined by arity
  - Performs fixed function on parameters
- Variable and constant nodes simply look up their value by index.
- Function nodes either recurse or push and pop values onto stack array.

Pro Tip: Check for stack overflow on SPU, particularly in debug builds.

Applications

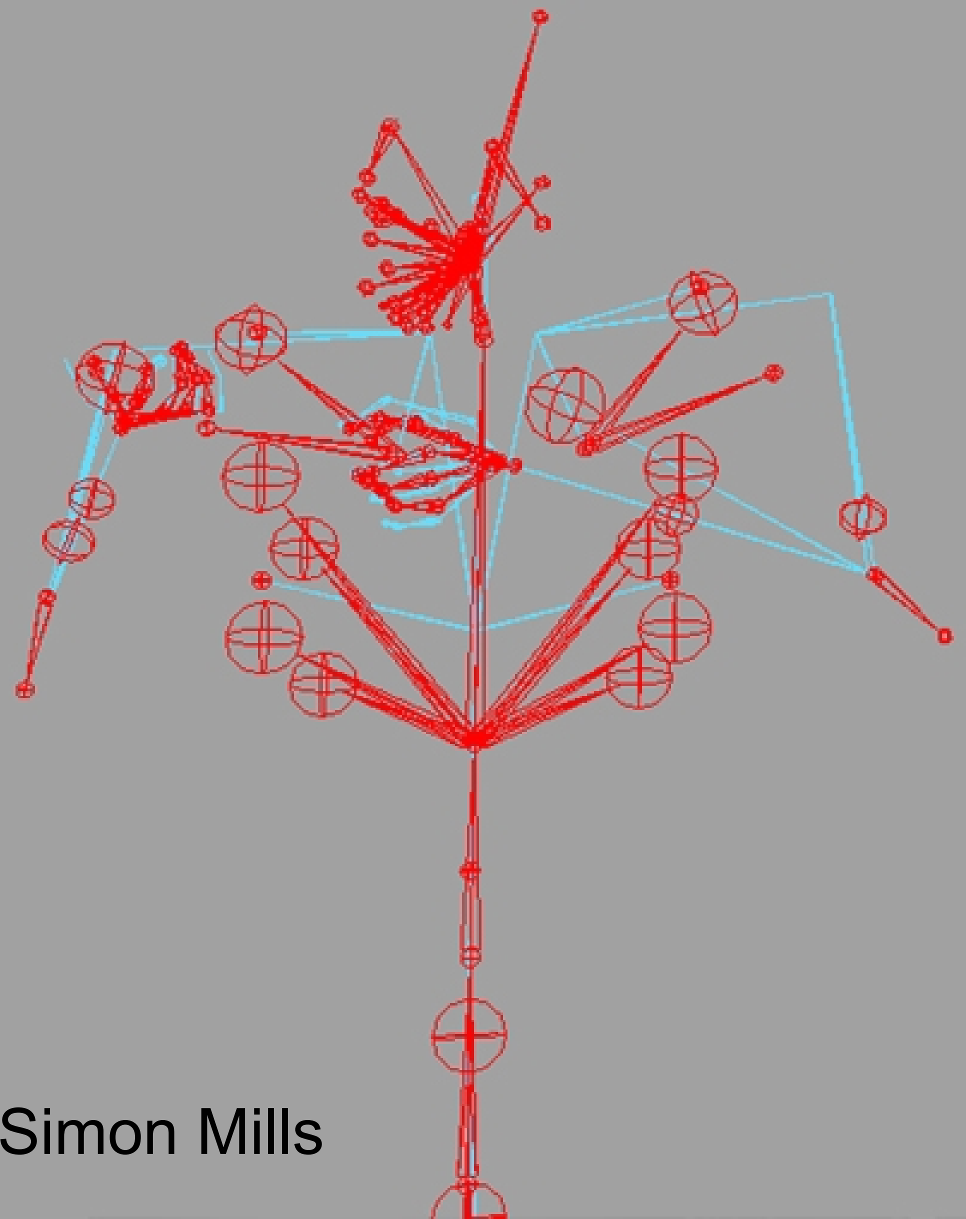
Unique Character Features

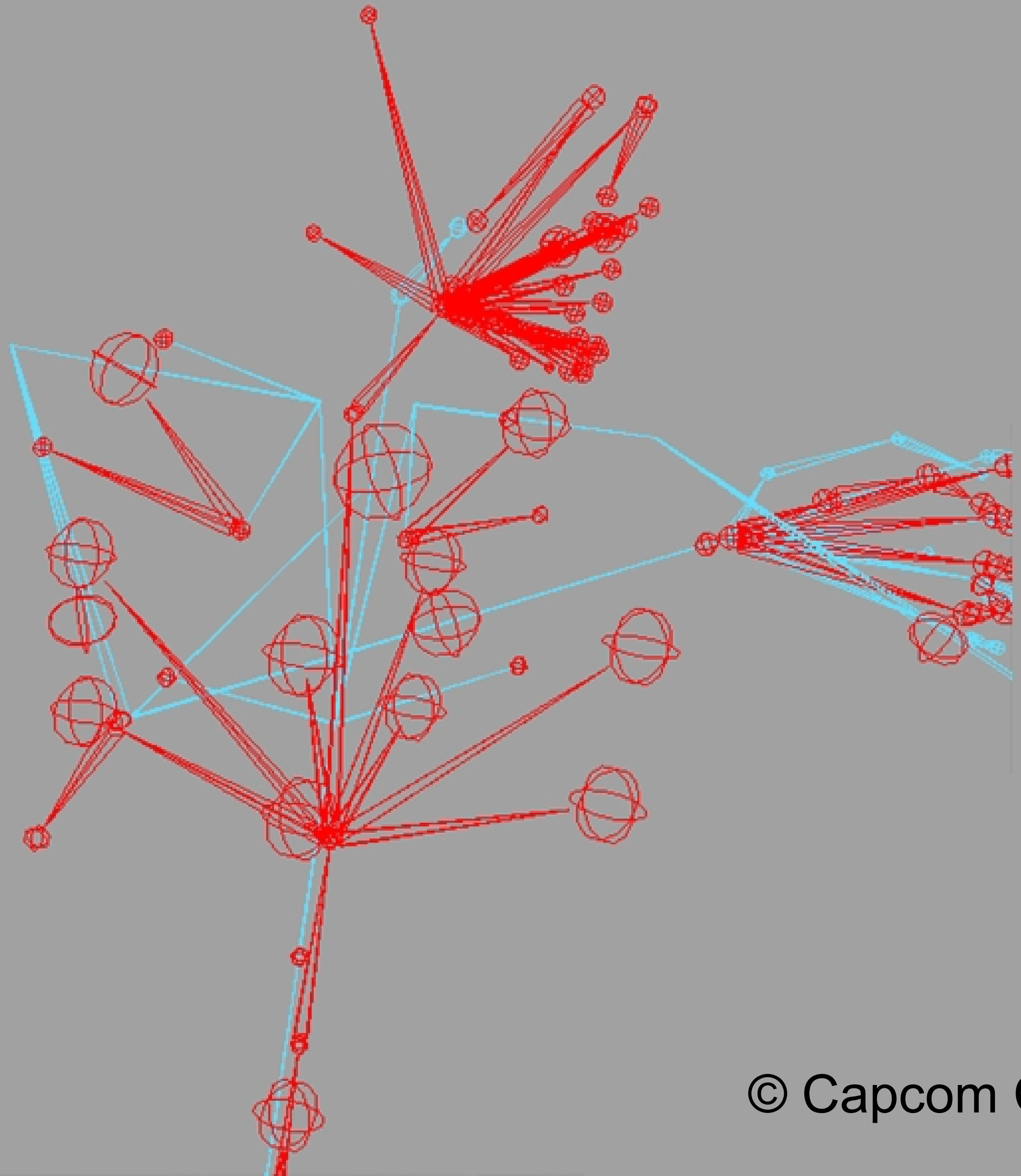
- Rigid knee and elbow pads.
- Improved seams.
- Beltway's robotic leg.
- Constraints for skirts and collars – great for mocap!
- Boss weak spot reveal animations controlled by single bone rotation.

Retargeting

- Playable characters in RE:ORC share animations.
- Animators work on a single standard rig.
- Female character rigs mostly driven by expressions.
- Meshes skinned to helper bones.
- Used to adjust height, shoulder width, arm and leg length.
- Saved **~10MB** RAM and **~18** man-months of animator time for retargeting and maintaining female animations alone.

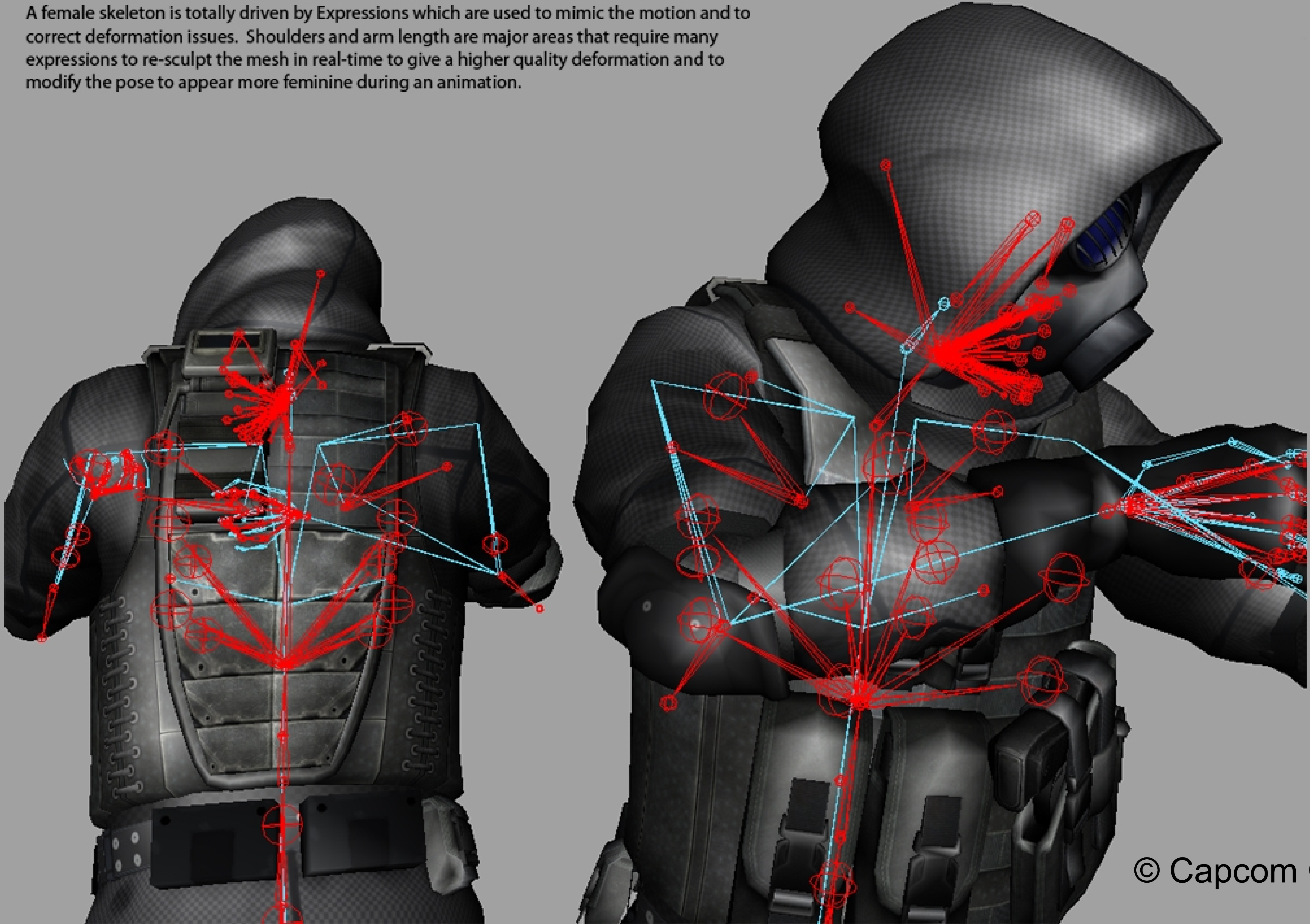
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



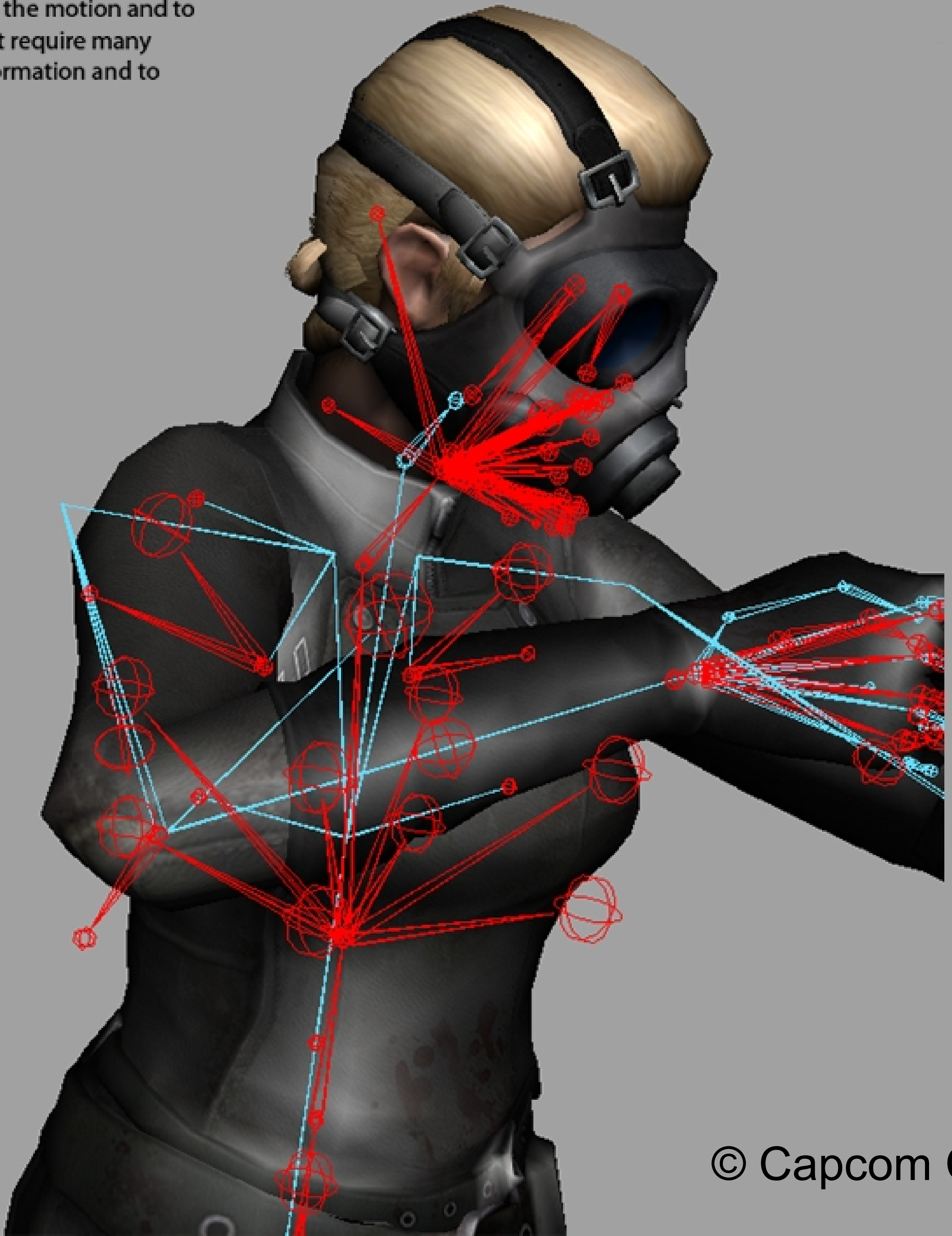


Rigging by Simon Mills

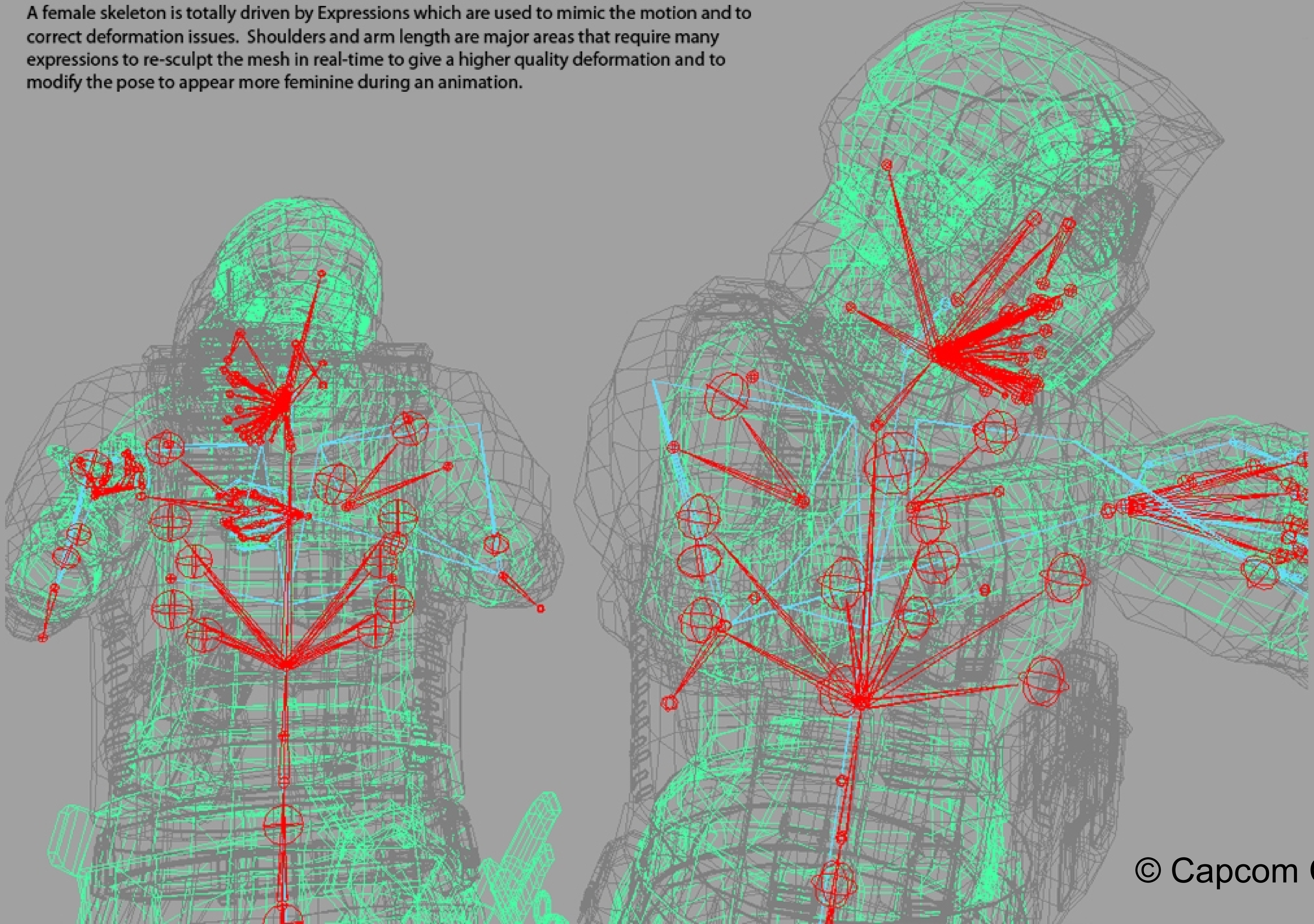
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



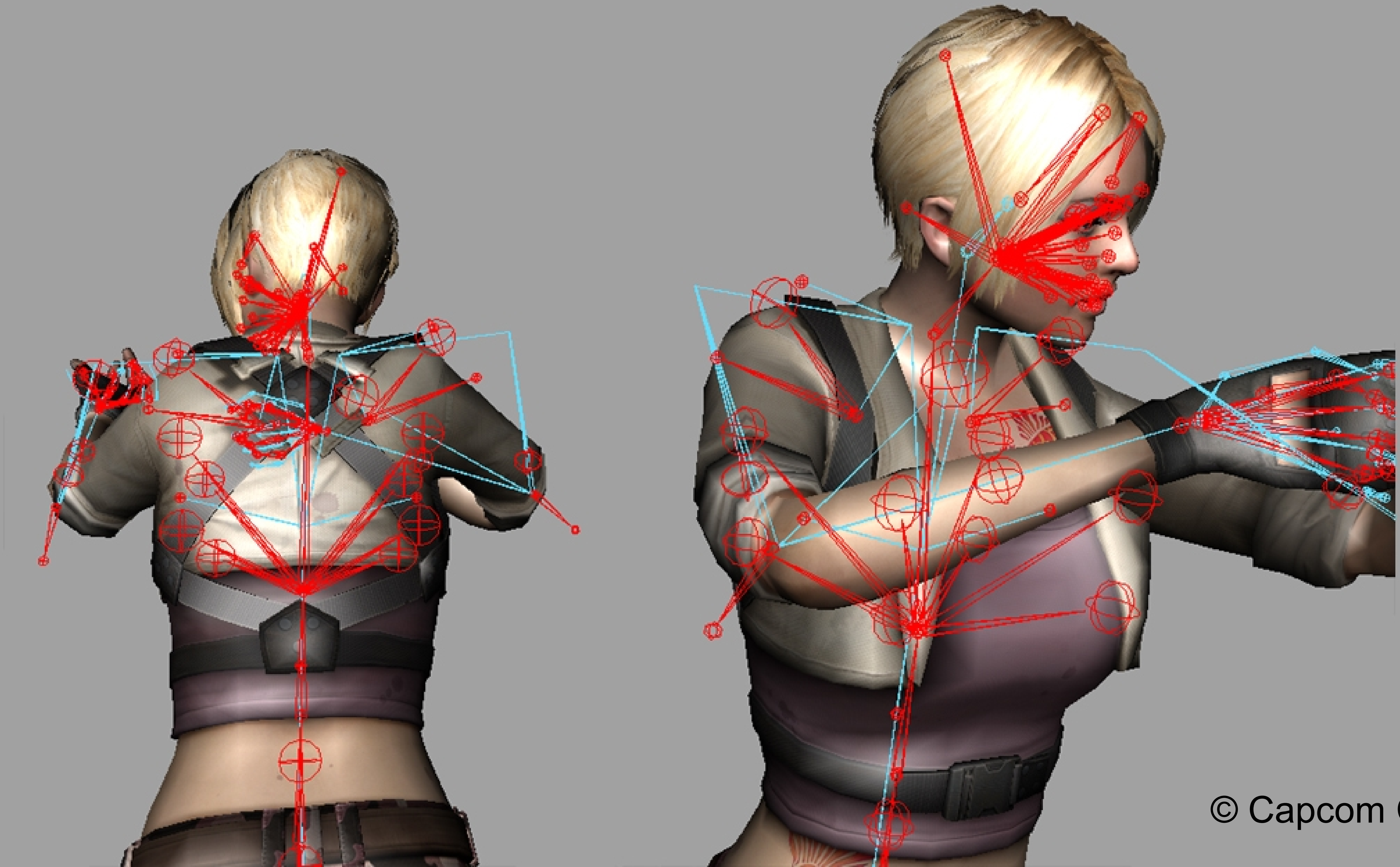
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



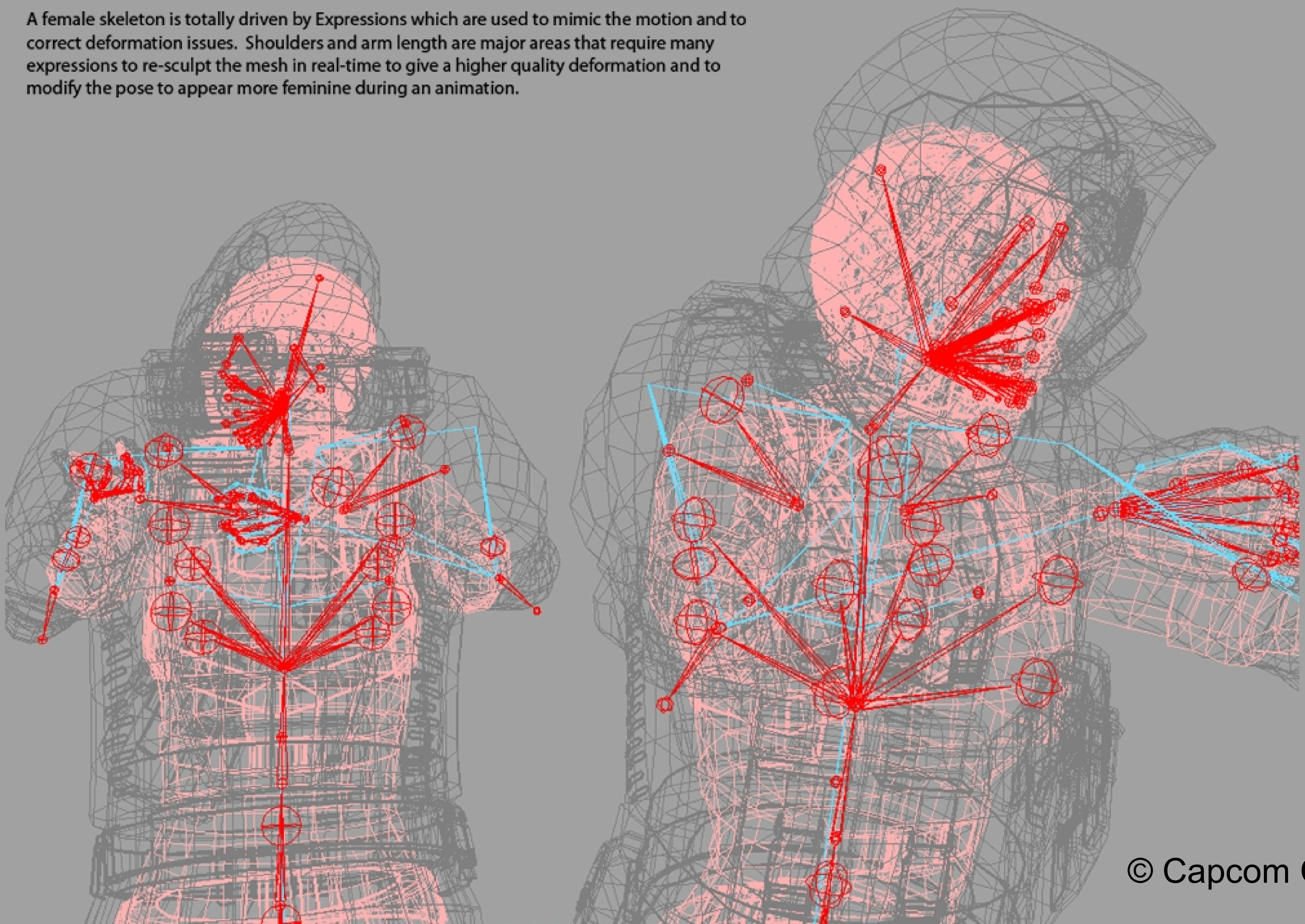
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



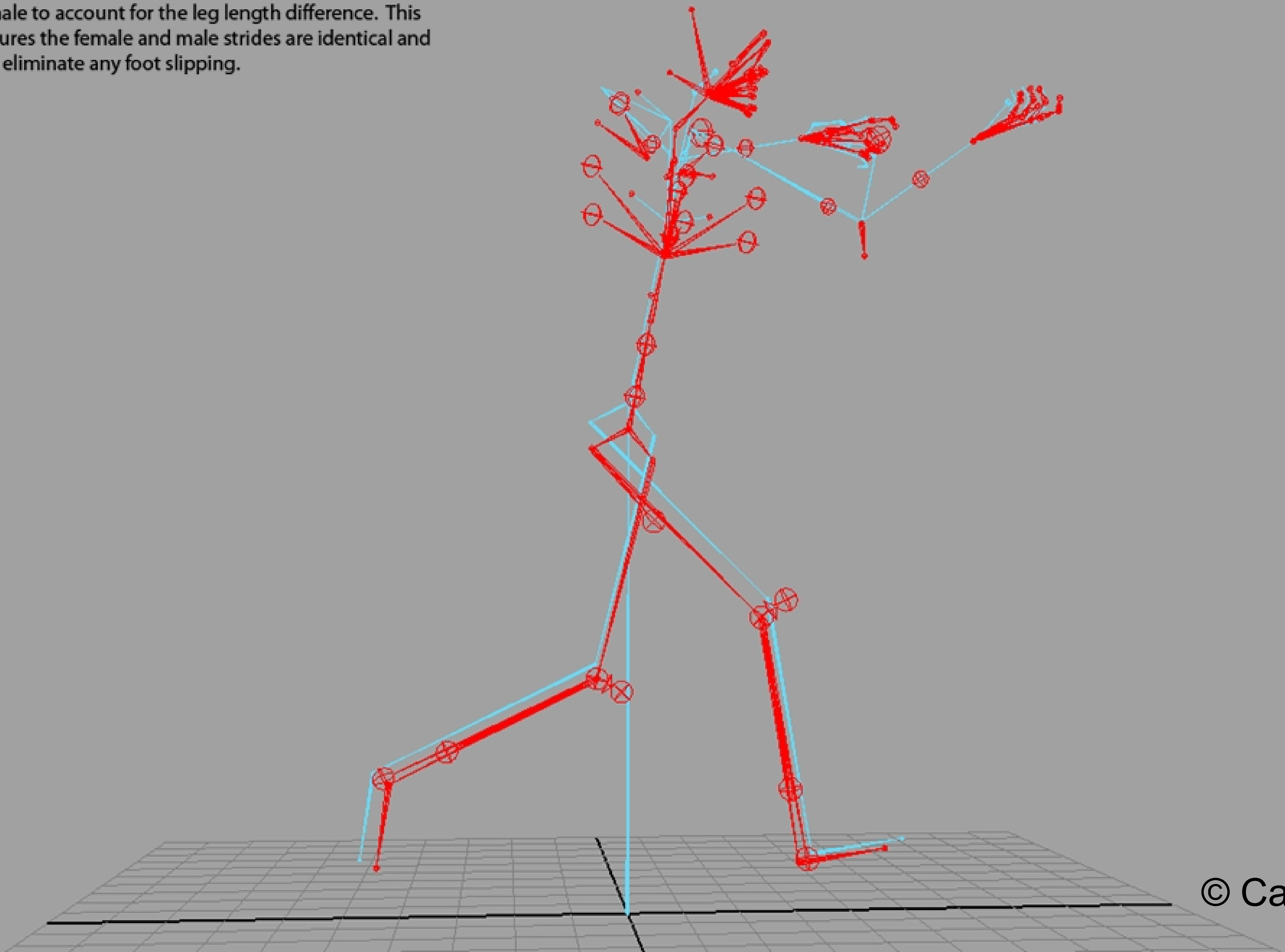
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



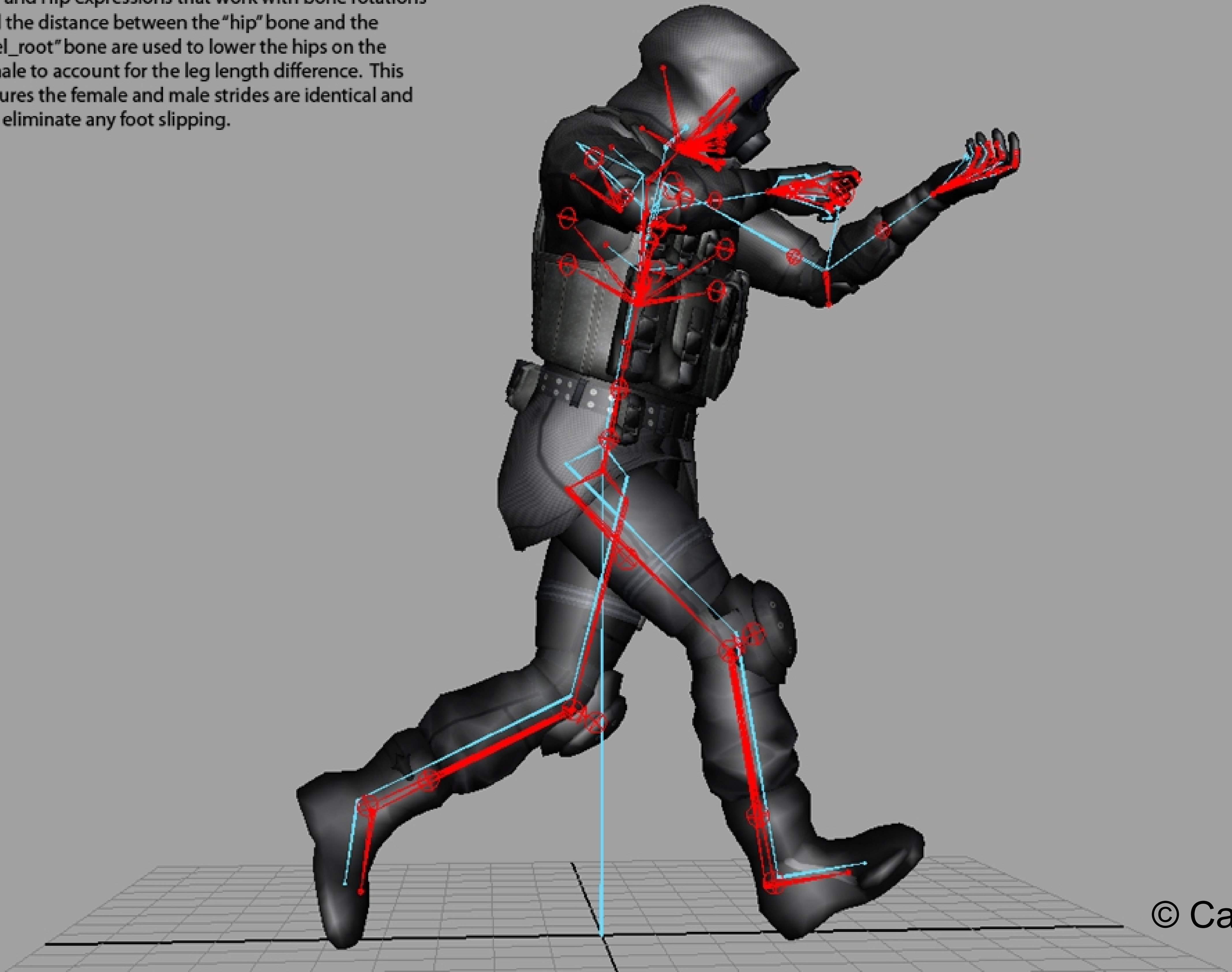
A female skeleton is totally driven by Expressions which are used to mimic the motion and to correct deformation issues. Shoulders and arm length are major areas that require many expressions to re-sculpt the mesh in real-time to give a higher quality deformation and to modify the pose to appear more feminine during an animation.



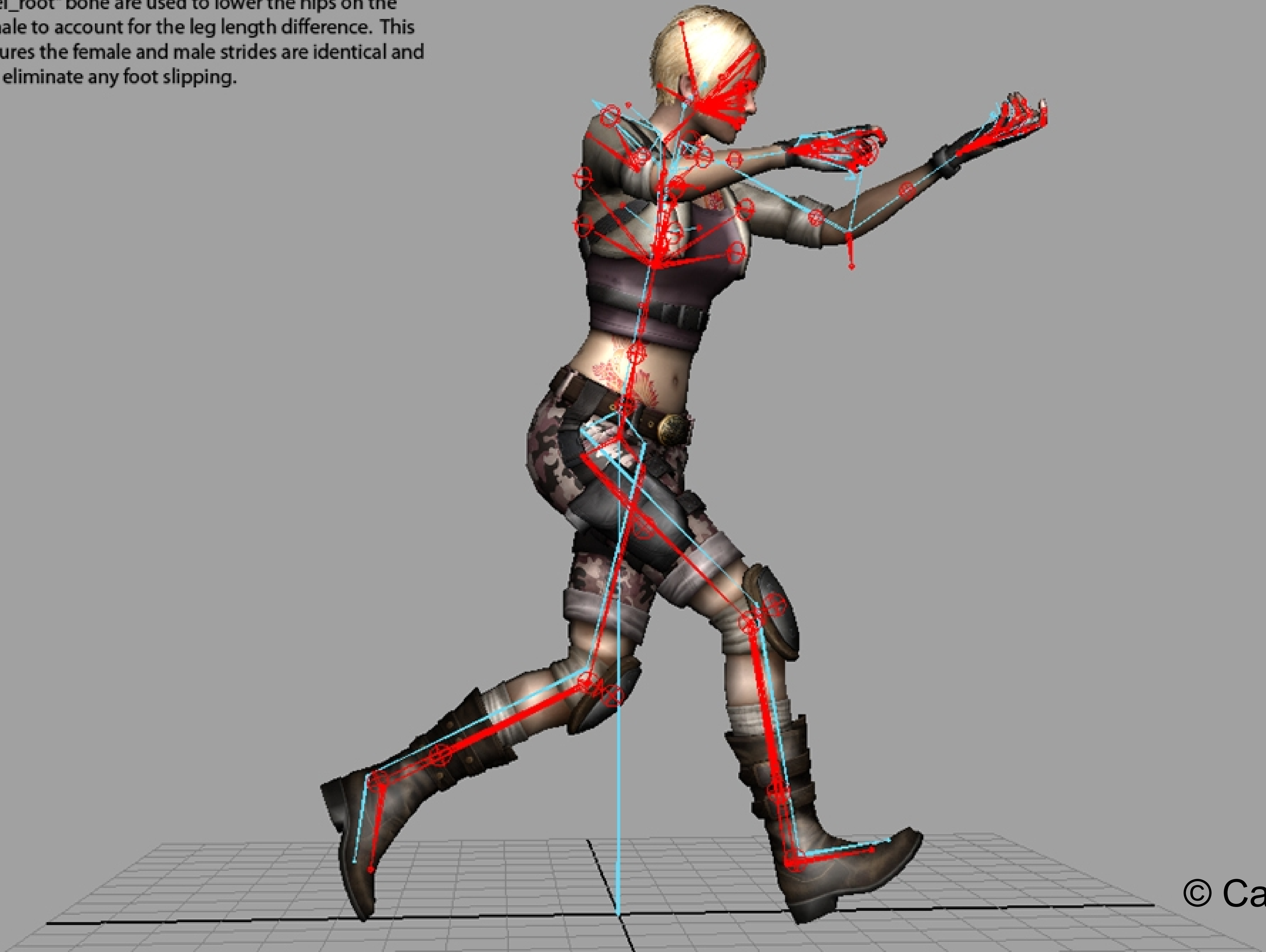
Leg and Hip expressions that work with bone rotations and the distance between the "hip" bone and the "skel_root" bone are used to lower the hips on the female to account for the leg length difference. This ensures the female and male strides are identical and will eliminate any foot slipping.



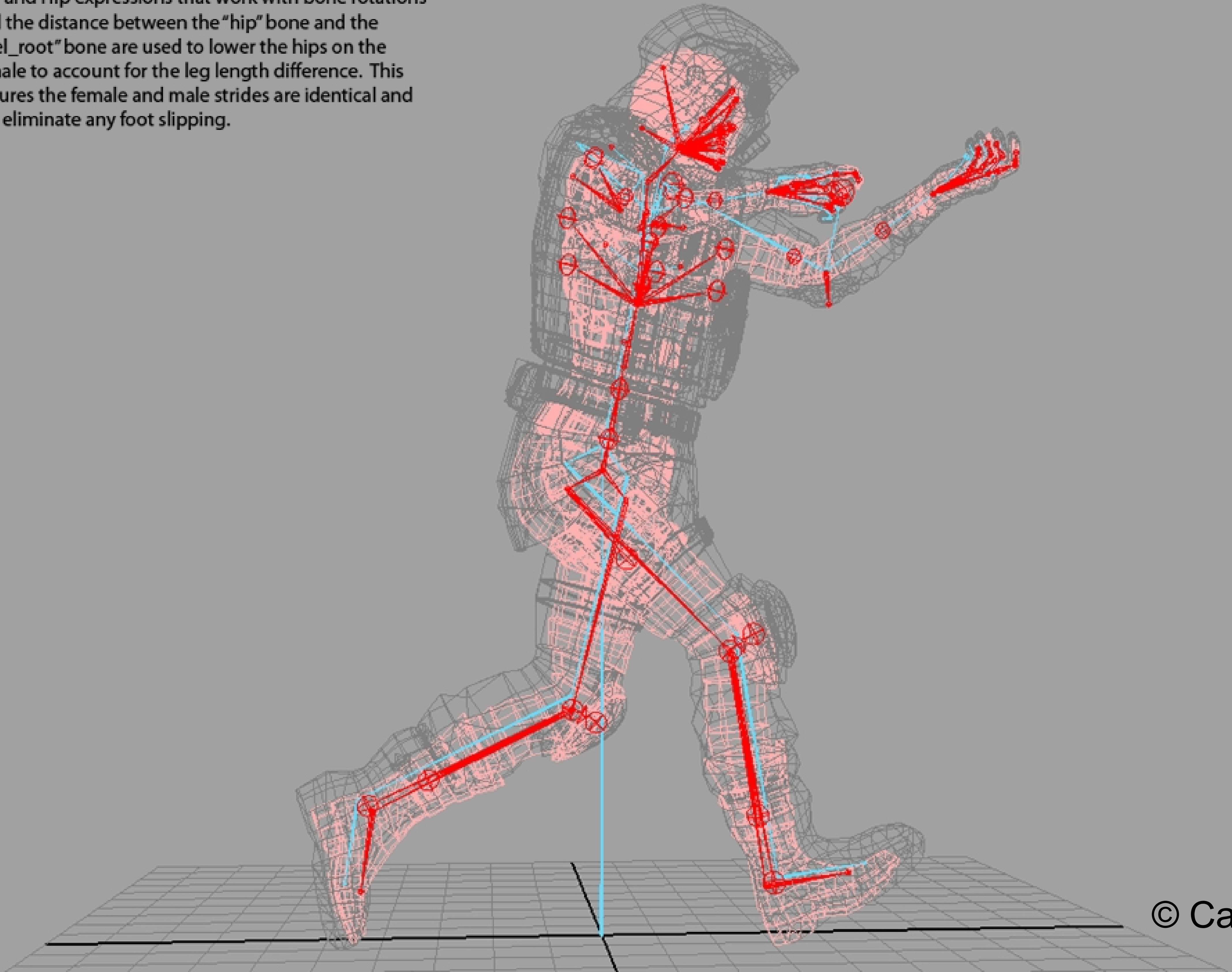
Leg and Hip expressions that work with bone rotations and the distance between the "hip" bone and the "skel_root" bone are used to lower the hips on the female to account for the leg length difference. This ensures the female and male strides are identical and will eliminate any foot slipping.

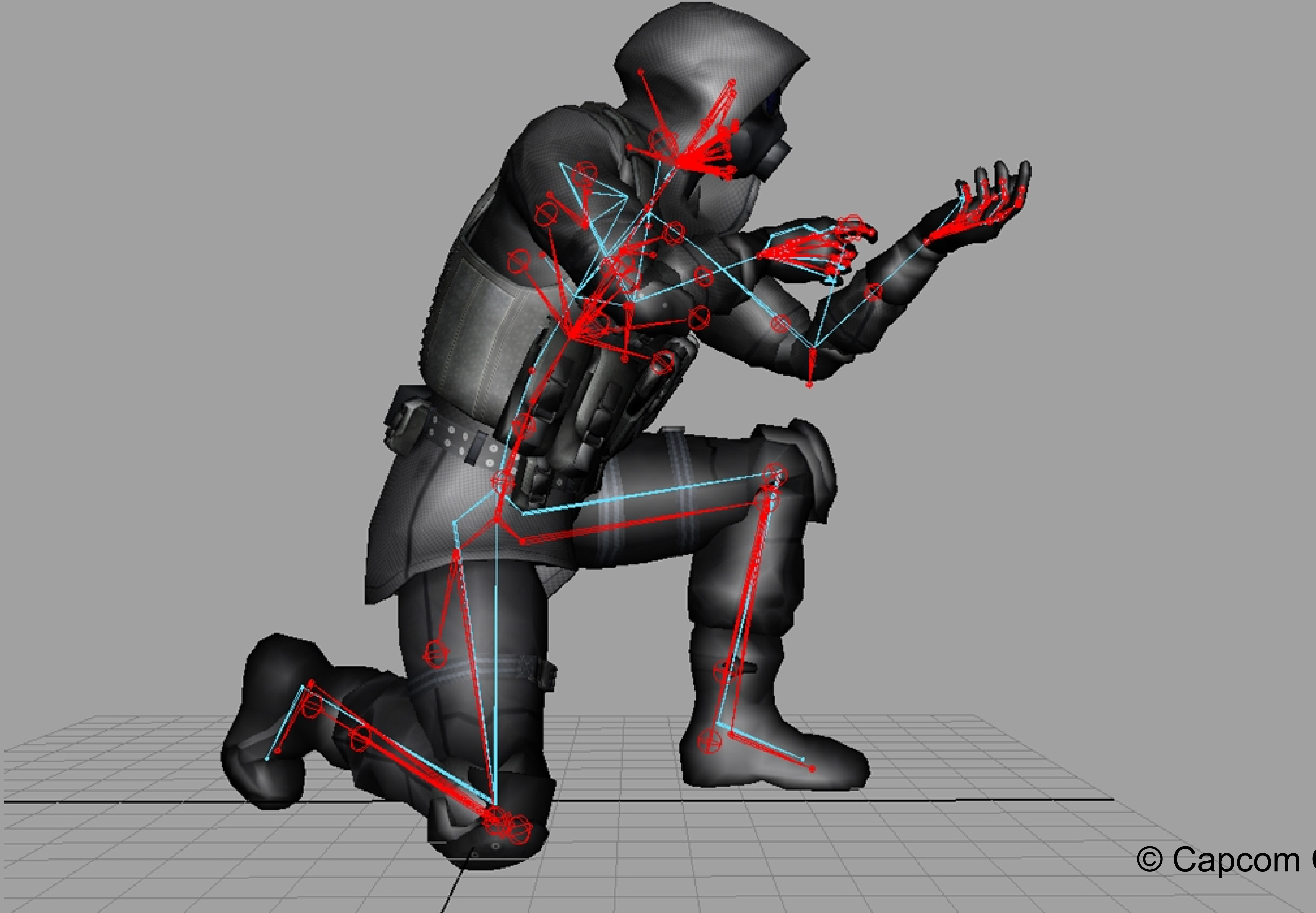


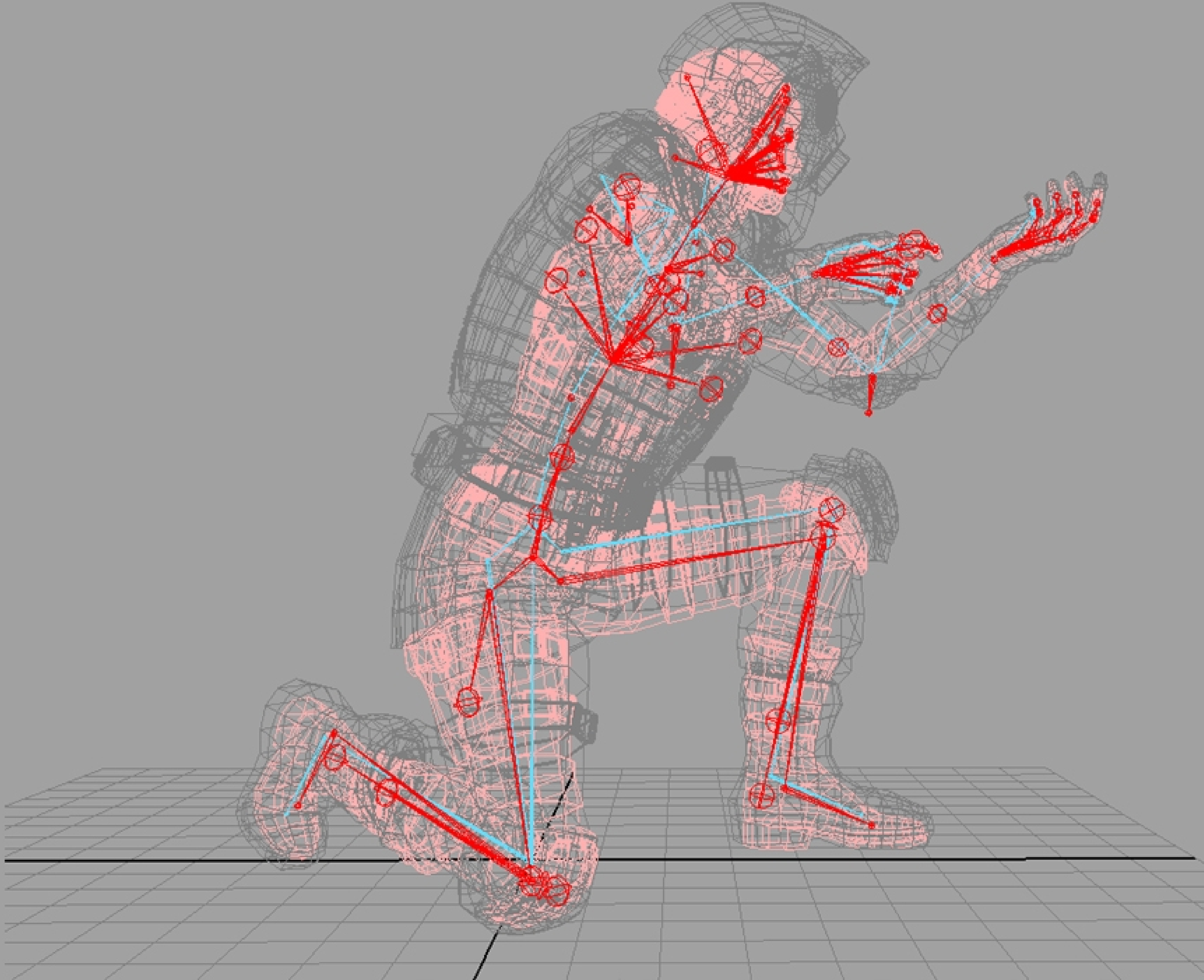
Leg and Hip expressions that work with bone rotations and the distance between the "hip" bone and the "skel_root" bone are used to lower the hips on the female to account for the leg length difference. This ensures the female and male strides are identical and will eliminate any foot slipping.



Leg and Hip expressions that work with bone rotations and the distance between the "hip" bone and the "skel_root" bone are used to lower the hips on the female to account for the leg length difference. This ensures the female and male strides are identical and will eliminate any foot slipping.







Advantages

- Decouples character art and animation. workflow. Maya / Motion Builder.
- WYSIWYG between Maya and engine.
- Works with procedural anim, e.g. IK, ragdoll, targeting.
- Streamed mocap data can be applied to characters interchangeably.
- Completely stateless.
- Very stable, especially with no division.
- Extremely lightweight and fast.
- Easy to hide with asynchronous jobs.
- Negligible GPU cost.

Disadvantages

- Female rig retargeting approach not physically based.
- Can cause some problems with IK and fully extended limbs.
- Limited to local space calculations.

Stats

- 45 characters use expressions in our game.
- Player animation skeleton has 68 bones.
- Average render bone count: 147 (Maximum: 176)
- Average expression count per character: 134 (Maximum: 255)
- Average binary file size: 10kB (Maximum: 14.2kB for Claire Redfield)
- Average bytecode length per expression: 5.46 (Maximum: 65)
- Average unoptimized bytecode length: 6.2 (Maximum: 77)
- Average SPU time per character (Optimized: 114.25μs, Unoptimized: 122.5μs)
- Average PS3 PPU time per character: 5μs (Job setup)
- Average Xbox 360 time per character: 343μs

Future Work

- Live pose transfer between Maya and engine for debugging.
- Add pipeline support for multiple compare/select branches.
- Lazy evalation of branches at runtime.
- Lazy conversion of pose transforms to Maya space.
- Eliminate code branches from evaluation loop (assembly/vector intrinsics)
- Write skinning transforms to command buffer hole / texture directly from job.
  - This will remove need for temporary render pose array storage, average ~10kB per character.
- Look at adding other useful built-in functions such as Min / Max.
- Support More advanced rigging features from Maya such as spline IK.

Questions?

If you have questions about this talk:

bhanke@slantsixgames.com

If you are interested in working with our game engine:

info@slantsixgames.com

All rigging in this presentation, including female rig solution:

Simon Mills, smills@slantsixgames.com

SLANT SIX GAMES