Programming Animation Using Behavioral Programming

David Harel(✉) and Shani Nitzan

Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, 76100 Rehovot, Israel
dharel@weizmann.ac.il, Shani.Lesser@gmail.com

Abstract. We propose a simple, user-friendly way of creating computer programs for hybrid systems whose execution involves animation. This is done by adapting *behavioral programming*, a recently proposed approach to software development that is aligned with how people describe system behavior, for use in programming animation. Users can define discrete and continuous behavior, which are then run simultaneously, interacting with each other, and resulting in a smooth hybrid animation.

1 Introduction

We define a natural and intuitive method for programming animation through scenarios. Each scenario describes a certain part of the motion of an object, and can correspond to an individual requirement, specifying what can, must, or may not happen following a sequence of events. Ideally, such motion scenarios should enable incremental development, allowing the user to add new scenarios without interfering with existing ones.

Behavioral programming (BP) is a recently proposed scenario-based programming paradigm that centers on natural and incremental specification of behaviors; see [8]. The BP approach was preceded by the language of *live sequence charts* (LSC) [1], which extends message sequence charts (MSC), and is a scenario-based language for reactive systems. LSCs add modalities to MSCs, allowing the specification of liveness and safety properties, as well as forbidden behaviors [1,2]. Two support tools for LSCs have been built, first the *Play-Engine* [2] and then *PlayGo* [3]. Later, the ideas where extended and embedded also in conventional programming languages like Java (resulting in BPJ, for behavioral programming in Java) [4], C++ [5], as well as Erlang and Blockly [6,7], thus providing a more classical programming point of view to this concept. See [8] for more details. In this paper we use BPJ.

Heretofore, behavioral programming had been used predominantly for programming discrete systems. In this article we propose and demonstrate its use for programming hybrid systems, whose execution can involve also continuous animation.

Animation programs can be executed by calculating the location of an object according to the time elapsed between clock ticks, while considering the location of the other objects. Although this can be made to produce satisfactory visual

E. Roubtsova et al. (Eds.): BM-FA 2009-2014, LNCS 6368, pp. 113–132, 2015.
DOI: 10.1007/978-3-319-21912-7_5

results, it is not always a natural way to describe animation. Over the years, much research has been carried out to simplify this by having moving objects assume more life-like behavior. Scenarios have been used for animation [9], and an initial attempt at using LSCs to create animations appears in [10]. A decision network framework for specifying and activating human behaviors has been introduced to create the behavioral animation of virtual humans [11].

Complex behavioral animation can also be obtained by defining simple, local rules between the various objects [12], which can rely on the objects having *synthetic vision* [13,14]. Synthetic vision has been used with other interaction components (an attention component, a gaze generation component and a memory component) to create a virtual human animation [15]. It has been integrated with cognitive science work on human locomotion to model interactive simulation of crowds of individual humans [16]. Non-linear dynamical system theory, kinetic data structures, and harmonic functions have also been used for agent steering and crowd simulation [17]. Motion within a large crowd has also been modelled by integrating global navigation and local collision avoidance [18].

A hybrid system is usually defined as one that exhibits both continuous and discrete (reactive) behaviors. Continuous behaviors are those that create the motion of an object; discrete behaviors are those that control sudden changes in that motion. Thus, between bounces, a bouncing ball exhibits a continuous behavior, its movement determined by angle, velocity, speed and gravity, but when it hits the ground it undergoes a sudden change in motion, due to the hit and the release of energy, which is considered a discrete behavior.

Hybrid behaviors can be modelled in many ways, a well known one being hybrid automata [19]. A hybrid automaton is a finite-state machine, where each state can be governed by a set of differential equations, enabling continuous behaviors between discrete state changes.

In this article we introduce a system that integrates defining local rules between various objects that have synthetic vision, with the *behavioral programming* principles. Our method simplifies the creation of animation in several ways, notably in that it enables the implementation of the local rules by using different threads for different rules. The method of synchronization of these threads is built into the BP execution mechanism, and in a way is transparent to the user. This is explained later.

A bouncing ball can thus be modeled using two scenarios: **Move** and **ChangeDirection**. The scenario **Move** is responsible for moving the ball according to its initial velocity and the force of gravity, and represents a continuous behavior. When the ball hits the ground, **ChangeDirection** calculates the new velocity of the ball, and it represents a discrete behavior. These scenarios can be implemented incrementally, so that for each step the physical correctness can be verified and simulated.

The code for all the examples presented in this article can be downloaded from:
https://www.dropbox.com/sh/yjxgpuq1pjzb2xd/AAAozws8Ql7l8ssAqmpKhmdAa?dl=0

2 Behavioral Programming

2.1 Basic Idioms

A behavioral program [8] consists of separate behavioral components (called *b-threads*) that generate a flow of events via an enhanced publish/subscribe protocol, as follows (see Fig. 1). Unlike regular threads, each b-thread runs atomically until it reaches a synchronization point, at which point it yields. When synchronizing, a b-thread specifies the following three sets of events:

- *Requested events:* The b-thread asks that these events be triggered, and to be notified when any of them is.
- *Waited-for events:* The b-thread asks to be notified when any of these events is triggered. It does not ask to trigger them.
- *Blocked events:* The b-thread prevents these events from being triggered.

When all the b-threads enter a synchronized point, some event is sought, which is in the requested events set of at least one of the b-threads, and is not in the blocked events set of any of the b-threads. One such event (if it exists) is selected for triggering, and when it is triggered all b-threads that requested or waited for it are notified, and their execution is resumed. Each of the resumed b-threads then proceeds with its execution, all the way to its next synchronization point, where it again presents its sets of requested, waited-for and blocked events. When all b-threads are again at a synchronization point the event selection process repeats. When more than one event is found as a legal candidate for triggering, i.e., it is requested by some b-thread and not blocked by any, the actual event to be triggered is chosen depending on the implemented execution semantics, of which there are several, see [8].

Each b-thread has local variables and global variables; events are defined globally. Since b-threads run atomically until a synchronization point is reached there is no need for safety measures, such as locks, to be taken in order to ensure that the b-threads work as expected.

BP principles can be implemented as part of various languages and programming approaches, with possible variations on the actual programming idioms. In addition to Java with the BPJ package, the idioms have been implemented in Erlang [6,20], Blockly [7] and C++ [5]. The BP idioms have also been applied to PicOS, a programming environment for wireless sensor networks, using C [21]. In addition to the Play-Engine and PlayGo, a visual approach called *Synthesizing Biological Theories* (SBT) [22], a tool for biological modelling, was implemented using the BP principles.

2.2 Behavioral Programming in Java

BPJ is implemented using the special **BPJ package**. In BPJ, every b-thread is an instance of the class `BThread`, and events are instances of the class `Event` or classes that extend it. The logic of each behavior is coded as a method supplied

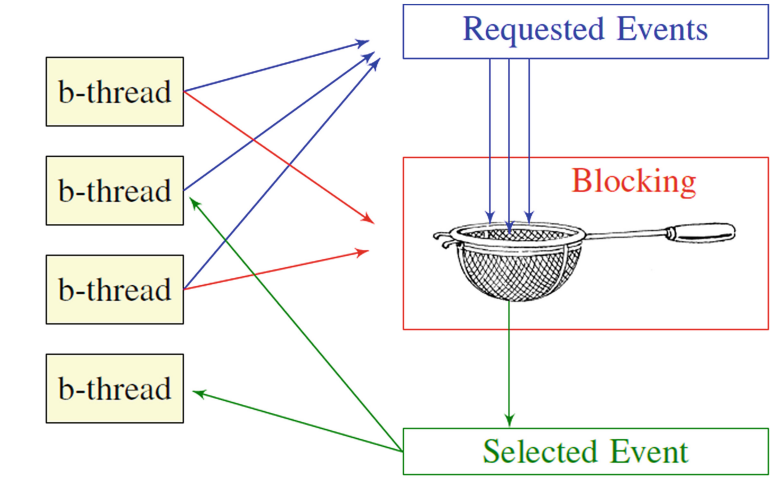


Fig. 1. An illustration of the BP execution cycle

by the programmer, which in turn invokes the method `bSync` to synchronize with other behaviors, and to specify its requested, waited-for and blocked events, as follows:

```
bSync(requestedEvents, waitedForEvents, blockedEvents);
```

When a b-thread calls `bSync`, it is suspended, and is resumed when a requested or waited-for event is triggered. To enforce predictable and repeatable execution the events selected at a synchronization point must be uniquely defined. This can be done in different ways. In BPJ every b-thread has a unique priority. When more than one event is requested and not blocked the event that will be triggered is the one requested by the b-thread with the lowest priority. If this b-thread requests more than one event the first event in the list of requested events that is not blocked is the one triggered (this is possible because in BPJ the requested events set is ordered).

Example 1. We illustrate the use of BPJ by a water flow control example taken from [4]. The goal is to have lukewarm water flow from a tap, by alternately letting a small amount of warm water and then a small amount of cold water

flow. Hot and cold water are supplied by different sources, each of which supplies its type of water repeatedly. The alternation of the two is done by an external mechanism. The b-threads for the three relevant behaviors are as follows:

1. **AddHotThreeTimes:** This b-thread requests the event `addHot` three times. The event `addHot` represents opening the hot water-tap for a short time.
2. **AddColdThreeTimes:** This b-thread requests the event `addCold` three times. The event `addCold` represents opening the cold water-tap for a short time.
3. **Interleave:** This b-threads repeatedly waits for the event `addHot` and blocks the event `addCold`, and then waits for the event `addCold` and blocks the event `addHot`.

AddHotThreeTimes and **AddColdThreeTimes** can work independently of one another, resulting in the flow of only hot or cold water from the tap. To get lukewarm water, the b-thread **Interleave** is used to force the alternation of the events `addHot` and `addCold`, which results in lukewarm water (Fig. 2).

The BPJ package, the code of the water flow problem and other BPJ examples can be downloaded from:
http://www.wisdom.weizmann.ac.il/~bprogram/bpj/

2.3 Live Sequence Charts

The visual language *live sequence charts* (LSC) [1] is an extension of message sequence charts (MSC). Like MSC, LSC use vertical lifelines to represent objects and horizontal arrows to represent messages passed between them. Since time flow is from top to bottom, a partial order of occurrences of the events ensues.

However the partial order alone cannot express what scenarios are to be carried out and when. This is where the extension of LSCs comes into play. LSCs can express what must happen (hot), what may happen (cold), and what is not allowed to happen (forbidden). Scenarios that are to be executed proactively are also distinguished from those that are only to be observed and monitored.

An LSC is composed of two parts, a prechart, depicted as a dashed-line hexagon, and a main chart. The main chart contains instructions that should be executed, and to activate it the scenario in the prechart must have occurred. The chart in Fig. 3 is a part of an implementation of a cruise control. It represents the scenario of the brake being pressed, resulting in the cruise releasing control of the brakes and the accelerator and turning itself off.

Play-in. LSC allows a new way of coding, called play-in [2], which is similar to programming by example. With play-in the user specifies the scenario in a way that is close to how real interaction with the system occurs, and the programming itself is done via a GUI. In the example in Fig. 3, the user would click the brake (action of the prechart), which releases the control the cruise has on the brakes and accelerator (action of the main chart).

```
class AddHotThreeTimes extends BThread {
    public void runBThread() {
        for (int i = 1; i <= 3; i++) {
            bp.bSync( addHot, none, none );
        }
    }
}

class AddColdThreeTimes extends BThread {
    public void runBThread() {
        for (int i = 1; i <= 3; i++) {
            bp.bSync( addCold, none, none );
        }
    }
}

class Interleave extends BThread
    public void runBThread() {
        while (true) {
            bp.bSync( none, addHot, addCold );
            bp.bSync( none, addCold, addHot );
        }
    }
}
```

Fig. 2. The water flow problem

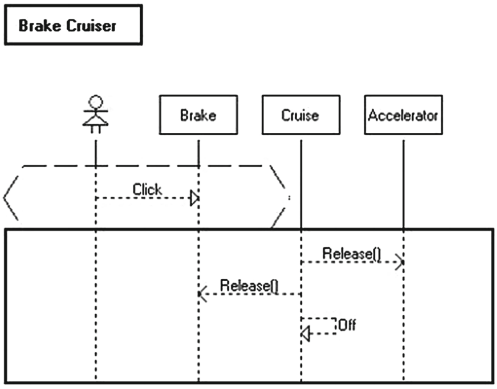


Fig. 3. LSC chart

Play-out. The play-out technique facilitates the execution of an LSC. Play-out does this by tracking the actions taken by the user and the system's environment. Events that may be selected next in all lifelines in all charts are tracked. When needed, play-out responds to an action accordingly, by selecting and triggering events. Play-out is also carried out via a GUI.

3 Animation in Behavioral Programming

Our method for creating animation with behavioral programming calls for each object to have b-threads that control its motion. Each of these b-threads represents

a certain behavior of that object and they interact with one another through the BP synchronization mechanism. The b-threads are divided into those that represent discrete behaviors and those that represent continuous behaviors.

B-threads that represent discrete behaviors are called *control b-threads*. They influence *motion b-threads*, those that represent continuous behaviors, through synchronization. After a synchronization point, when a control b-thread senses that the motion of the object it is in charge of needs to be manipulated it will do so.

It should be noted that in this article continuous behaviors are implemented by discretization (the process of transforming a continuous model into discrete parts), since computers work in a discrete way. This can be analogous to a person walking. Even though the person is moving in a continuous way his/her motion can be divided into discrete steps.

Like in regular animation programs, small movements of an object are triggered by time ticks, which are controlled by the following b-thread:
`Sleep(Event endSleep, long timeOfSleep, double priorityOfBThread)`

When created, this b-thread sleeps for `timeOfSleep` milliseconds and then requests that the event `endSleep` be triggered. The b-thread gets priority `priorityOfBThread`. The b-thread `Sleep` is always created by a motion b-thread.

3.1 Motion B-Threads

Since motion b-threads represent an object's continuous behavior, and continuous behavior in animation is guided, among other things, by the velocity of an object, the most basic and simple pattern of a motion b-thread is an infinite loop that does the following: It first creates the b-thread `Sleep`, and then waits for the event `endSleep`. It then requests the event `takeStep`, and finally calculates the new position of the object.

while *true* **do**
  Create the b-thread **Sleep**.
  Wait for the event `endSleep`.
  Request the event `takeStep`.
  Calculate the new place of the object according to the velocity and the time passed.
end while

Algorithm 1. Basic motion b-thread algorithm

To understand how a motion b-thread works, imagine a green ball rolling to the right. The ball has one continuous behavior, which is its movement. Its motion b-thread is as follows:

```
while (true) {
    createSleep(endSleep1, sleep, prio1);
    bp.bSync(none, endSleep1, none);
    bp.bSync(takeStep1, none, none);
    x1 += getTimePassed()*step; //updates coordinate
}
```

An example of this animation can be viewed here:
https://www.dropbox.com/s/jaoptmayfn1cm8c/Sec3Sub1.mp4.

Now suppose another ball is added; this time a blue ball rolling at half the speed of the green ball. Now there is a second continuous behavior, which is represented by another appropriate motion b-thread.

An example of this animation can be viewed here:
https://www.dropbox.com/s/56eoxjxwcybv3cb/Sec3Sub1_2.mp4.

With these two b-threads the two balls move simultaneously, each at its own pace. New moving objects can thus be added incrementally. Changing the direction or speed of one of the balls requires no change in the b-threads of the other balls.

3.2 Control B-Threads

Control b-threads can manipulate the motion of an object in different ways. In our example, one of the simplest is to block the event `takeStep`. The motion b-thread of the object being blocked terminates, which results in stopping the motion. Imagine the rolling green ball of the previous subsection, and that this ball is getting closer to a wall. When it reaches the wall we want it to stop rolling. The continuous behavior of the ball is the same, which means that the motion b-thread that represents it is the same as well, but now the following control b-thread can be added:

```
while(true){
    bp.bSync(none, endSleep1, none);
    if(collision()) { //checks if the ball has reached the wall
        bp.bSync(none, none, takeStep1);
    }
}
```

An example of this animation can be viewed here:
https://www.dropbox.com/s/oh5cxuex8ww8i2k/Sec3Sub2.mp4.

This b-thread adds a behavior without the user having to alter existing ones. More control b-threads can be added very easily. For example, all that needs to be done in order to add a new wall is to enhance the program with a new control b-thread that checks collision with the new wall and blocks `takeStep1` accordingly.

These basic algorithms for control and motion b-threads are the basis for implementing far more complex behaviors, as we show later.

3.3 Improving and Adding B-Threads

One problem that appears when integrating different b-threads for the balls, is that a ball does not stop exactly when it reaches the wall, but a little later. This is because the ball moves in steps and the wall will often be reached in

between two steps. In some cases this kind of issue does not cause a problem. For example, if an object moves until it sees an obstacle 50 m away and it turns left, then it does not matter if the object turned left when it was 50 m away from the obstacle or 50 m and 1 cm away. However, in many cases, including the one above, this does matter. The ball should appear to stop at the wall, which means that it has to stop exactly when it gets to the wall.

To overcome this problem we make the control b-thread look ahead at the next step, and if a change of motion is in order, it is fixed. This is done by making small changes to both the motion and the control b-threads. The changes are demonstrated in the bouncing ball example. The motion b-thread calculates the new y-axis coordinate of the ball, and then requests the event `takeStepY`. After it is triggered, the new coordinate is updated. The motion b-thread is as follows:

```
while (true) {
    createSleep(endSleep, sleep, prio);
    nextY = calcNextY(); //calculates the next coordinate
    bp.bSync(takeStepY, none, none);
    y = nextY; //updates the new coordinate
}
```

After `checkStep` (the event that symbolizes looking ahead at the next step) is triggered, the control b-thread checks if the next coordinates will result in a collision between the ball and the ground. If so, the b-thread requests the event `hitGround`, and updates the next coordinate so that the ball is exactly on the ground. Since the ball should bounce and not just stop, then instead of blocking the event `takeStepY` the control b-thread calculates the new velocity, which is in the opposite direction of the previous velocity, and its speed is slower, due to friction. The control b-thread is as follows:

```
while (true) {
    bp.bSync(none, endSleep, none);
    bp.bSync(checkStep, none, none);
    if(ballHitGround()){ //checks if the ball collides with the ground
        bp.bSync(hitGround, none, none);
        nextY = calcNewNextY(); //calculates the new next coordinate
        calcNewInitVelocityY(); //calculates the new velocity
    }
}
```

By using only these two b-threads the ball continues to bounce on the floor forever, reaching increasingly lower latitudes. To stop the ball when its velocity is near zero, another control b-thread is added. This one waits for the event `hitGround`, then blocks the event `endSleep` if the velocity is near enough to zero. The b-thread is as follows:

```
while(true) {
    bp.bSync(none, hitGround, none);
    /*The velocity is near zero so motion should stop*/
    if(speedNearZero())
        bp.bSync(none, none, endSleep);
    }
```

An example of this animation can be viewed here:
https://www.dropbox.com/s/gg2m18rg5erjhjc/Sec3Sub3.mp4.

4 Using Behavioral Programming for Billiard

Our technique for creating animations using behavioral programming is fairly simple, yet it can handle many of the general problems that occur when trying to program animations. We now discuss billiard. The animation scenarios of a billiard game are described in Fig. 4. The game consists of 16 balls; here the balls are indexed 0–15, where 0 is the white ball. The event `moveBall[i]` is triggered when ball number i starts moving (which happens when another ball hits it).

The b-thread that represents the continuous behavior of ball i is **MoveBall(i)**. Since the ball should only start moving when the event `moveBall[i]` is triggered, this b-thread repeatedly waits for it. After that, as long the velocity of the ball is not zero the ball should move while decreasing its velocity at each

1. **StartMove**: When a new move starts, the player can use the stick to hit the white ball.
2. **StickHitsWhiteBall**: When the stick hits the white ball, the white ball starts moving. Its velocity is calculated by the direction and impact of the hit.
3. **MoveBall**: As long as the velocity of the ball isn't zero, it moves and the velocity is decreased by friction. As long as the ball is moving a new move can't start.
4. **BallHitsBorder**: When a ball hits the border of the table, the ball's motion stops and it starts moving in the opposite direction.
5. **BallInHole**: When the ball enters one of the holes, it stops moving, disappears from the table, and other balls on the table should not consider it in their motion.
6. **WhiteBallInHole**: When a new move starts, if the white ball is inside a hole, it is put back in the middle of the table.
7. **BallHitsBall**: When balls collide, both balls stop moving and then start moving with a new velocity determined by the direction and impact of the hit.

Fig. 4. Description of the animation scenarios of a billiard game

time tick, due to friction. This is exactly what **MoveBall(i)** does, while blocking the event `startMove` with every call to the function `bSync`. Therefore, a new move does not start while the ball is still moving. **MoveBall(i)** is as follows:

```
while(true){
    bp.bSync(none, billiard.moveBall[i], none);
    long t1 = System.currentTimeMillis(), t2 = t1; //initialized the time

    /*continues in a loop until the ball's velocity equals zero*/
    while(ballHasVelocity()){
        createSleep(endSleep[i], sleep, prio[i]);
        bp.bSync(none, endSleep[i], startMove);

        t2 = System.currentTimeMillis(); //updates the time
        calculateNextStep(t2-t1); //calculates the next coordinates of the
              ball

        bp.bSync(takeStep[i], none, startMove);

        takeNextStep(); //updates the coordinates of the ball
        t1 = t2;
    }
}
```

There are three discrete behaviors in the animation of a billiard game: a ball's collision with the borders of the billiard table (**BallHitBorder(i)**), a ball falling into one of the holes on the table (**BallInHole(i)**) and a ball colliding with another ball (**BallHitsBall(i,j)**). Every step these b-threads wait for the event `endSleep[i]` (which is only requested when ball i has non-zero velocity).

Every time tick, **BallHitBorder(i)** checks if ball i has collided with the borders of the table and updates the next coordinates of the ball and its velocity accordingly. **BallInHole(i)** does the same for the holes of the table, with one exception: when the ball falls into a hole it should stop moving altogether and other balls shouldn't check for collisions with it. To insure this as long as ball i is inside the hole, the event `ballOnTable[j]` is blocked.

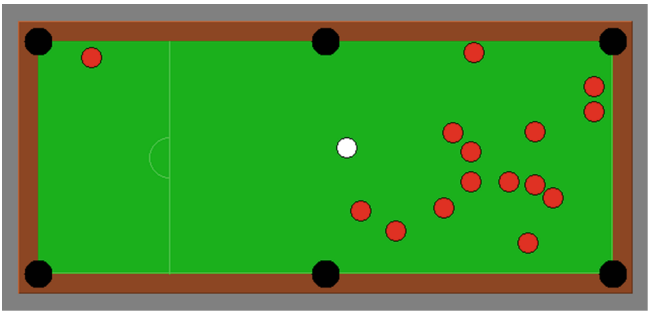
For every ordered pair of balls i and j, the b-thread **BallHitsBall(i,j)** represents ball i colliding with ball j. When there is a collision, the coordinates and velocities of both balls should change accordingly, and if ball j was motionless it should start moving. To enable this, every time a collision is detected, the event `moveBall(j)` is requested, which resumes the execution of the motion b-thread **MoveBall[j]**. After checking for a collision **BallHitsBall(i,j)** requests the event `ballOnTable[j]`, which prevents this b-thread from continuing if ball j is not on the table. **BallHitsBall(i,j)** is as follows:

```
while(true) {
    bp.bSync(none, endSleep[i], none);
    bp.bSync(checkCollision[i], none, none);

    //checks if there is a collision between balls i and j
    if(areBallsColliding(i ,j)){
        adjustPosition();
        adjustVelocities();

        bp.bSync(moveBall[j], none, none);
    }
    bp.bSync(ballOnTable[j], none, none);
}
```

All the control b-threads are independent of each other, while still affecting the motion b-thread when it is required. This is important, and makes it possible to add or remove discrete behaviors easily, without affecting other b-threads. The current example shows only one motion b-thread per object, which may not always be the case.

An example of this animation can be viewed here: https://www.dropbox.com/s/oukaccxcke2m26n/billiard.mp4.

The billiards animation was based on open source code that can be downloaded from here: http://ftparmy.com/193538-billard4k.html.



5 Adding Continuous Behaviors

Sometimes moving objects have more than one continuous behavior. When a ball is thrown it can be thought of having two continuous behaviors. The first is the motion in the direction the ball was thrown, and the second is the acceleration towards the ground, due to gravity. In our approach, like in many motion calculations, each of these behaviors can be represented by an independent motion b-thread, and integrating them is done using one or more control b-threads.

Separating an object's motion into multiple continuous behaviors simplifies the act of describing the motion. Describing a thrown ball with a single scenario is difficult, because the ball has a curve-like motion. When the motion is separated

into the behaviors of movement with the initial velocity and movement towards the ground due to gravity, describing the motion becomes easy.

Like with discrete behaviors, adding continuous behaviors to an already working program should be done, as far as possible, without altering existing b-threads. The BP's incremental approach makes adding continuous behaviors of an object relatively easy. Moreover, implementing motion b-threads that represent such continuous behaviors creates additional benefits. For example, it makes the controlling of the motion simple. If there are several continuous behaviors that work at different times, scheduling them can be done by control b-threads.

5.1 How it is Done

To demonstrate the b-threads of a thrown ball having an initial velocity parallel to the x-axis, we use the example of a bouncing ball, which already has continuous behavior of the acceleration due to gravity. Thus, the motion in the direction that the ball was thrown is the only behavior that needs to be added.

The new b-thread has the same pattern as other motion b-threads. Each time the event `endSleep` is triggered this motion b-thread requests the event `takeStepX`, and then updates the new x-axis coordinate of the ball. The b-thread is as follows:

```
while (true) {
    bp.bSync(none, endSleep, none);
    bp.bSync(takeStepX, none, none);
    double x = calcNewX(); //updates new x-axis coordinates
}
```

Notice that the main difference between this motion b-thread and the earlier ones is the fact that this one does not trigger the b-thread **Sleep**. This is because the motion b-thread in charge of the motion towards the ground already does this. Since we assume that every object has a single clock and all its b-threads work in a way that is aligned with that clock, only one motion b-thread triggers **Sleep**. We call it the *main motion b-thread*. All other b-threads (control and motion) just wait for the event `endSleep` when a synchronization with the clock is called for.

This scenario presents a problem that can occur when programming animations using behavioral programming. Sometimes a control b-thread can affect more than one motion b-thread even when this is not desirable. The fact that both motion b-threads work with the same clock means that, automatically, when `takeStepY` is blocked due to the y-axis velocity being near zero the ball stops moving along the x-axis too. If this scenario is not desired, then creating the b-thread **Sleep** should not be done by the motion b-thread that represents the motion on the y-axis. Rather, it can be done by creating a new main motion b-thread.

In our example, every time the ball hits the ground the direction of the y-axis velocity is flipped and its speed decreases due to friction. Although the

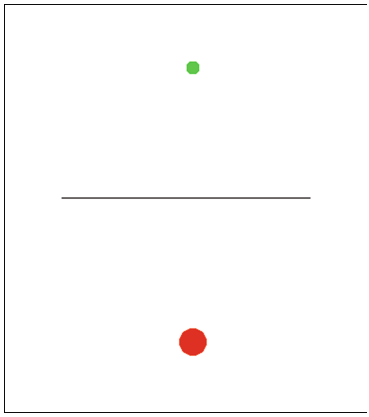


Fig. 5. Circle trying to pass an obstacle to get to its destination

x-axis velocity should not be flipped its speed should decrease. This can be done easily by adding a control b-thread that waits for the event `hitGround` and then decreases the speed of the x-axis velocity.

An example of this animation can be viewed here:
https://www.dropbox.com/s/ldy2wjv122s5lia/Sec5Sub1.mp4.

5.2 Blocking Unwanted Continuous Behaviors

When using more than one motion b-thread it is possible to block some of the events requested by some of those b-threads. Every time a specific motion is deemed unnecessary or harmful, an event requested by the motion b-thread representing it can be blocked by a control b-thread.

This is demonstrated by the following example, which involves a green circle that has to get to a destination point (depicted by a red circle), overcoming a mid-way obstacle (in the form of a line). The green circle has to first move to the closest edge of the obstacle and then move to the destination. There are three motion b-threads involved (Fig. 5):

1. **MoveY-** This b-thread adds one unit to the circle's y-axis coordinate every time tick, and is the main motion b-thread, since the circle's coordinates should be increased until it reaches its destination.
2. **MoveX1-** This b-thread adds one unit to the circle's x-axis coordinate every time tick.
3. **MoveX2-** This b-thread removes one unit from the circle's x-axis coordinate every time tick.

Since the circle is continuously moving forward on the y-axis, the b-thread **MoveY** should run until the circle reaches the destination. If the circle passes the

obstacle from the right-hand side, **MoveX1** should run until the circle reaches the obstacle. The b-thread **MoveX2** should start running only when the circle reaches the obstacle, and should stop when it reaches the destination.

In this example there is one control b-thread. As long as the circle has not reached the obstacle it blocks the event `moveX2`. After that, as long as the circle has not reached the destination, this b-thread blocks the event `moveX1`. When the ball reaches the destination it should stop moving, which is why `moveX1`, `moveX2` and `moveY` are blocked. The b-thread is as follows:

```
//continues until circle reaches the obstacle
while(!reachObstacle())
    bp.bSync(none, endSleep, moveX2);

//continues until circle reaches the destination
while(!reachDestination())
    bp.bSync(none, endSleep, moveX1);

//blocks all movement of circle
bp.bSync(none, none, new EventSet(moveY, moveX1, moveX2));
```

An example of this animation can be viewed here:
https://www.dropbox.com/s/kad7uicwipwqfls/Sec5Sub2.mp4.

5.3 Random Continuous Behaviors

Imagine a winding corridor, and suppose that one should get an object from one end of the corridor to the other. There are many known ways to get the object to its destination. Here we show how animation using behavioral programming can be used for this (Fig. 6).

The solution to this problem using BP is very simple. There is a set of directions D in which the object can move. These are defined ahead of time. For every direction $d \in D$ a motion b-thread (**Move**) and a control b-thread (**BlockMove**) are written.

The b-thread **Move** works in a different way from the motion b-threads presented so far. It waits for the event `takeStep(`d`)`, and then updates the coordinates of the object according to the time passed and the direction d. **Move** is as follows:

```
while (true) {
    bp.bSync(none, takeStep, none); //waits for takeStep(f)
    //updates coordinates according to the function f
    x += speedX;
    y += speedY;
}
```

Every time tick, **BlockMove** checks if moving the object in the direction d will result in the object being too close to the walls of the corridor, and blocks the event `takeStep(`d`)` if it does. Here is how this is programmed:

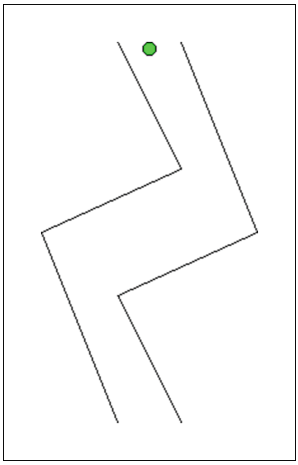


Fig. 6. An object that wants to get from one end of a winding corridor to the other

```
bp.bSync(none, endSleep, event);
while (true) {
    //checks if moving in the direction of function f results in
        intersection with the borders
    if(intersectWithBorders())
        bp.bSync(none, endSleep, takeStep); //blocks takeStep(f) until
            next time tick
    else
        bp.bSync(none, endSleep, none);
}
```

To trigger the event `takeStep(`d`)`, another b-thread is used, which is the main motion b-thread. It waits for a time tick and then arranges the events `takeStep(`d`)` for every direction $d \in D$ in a random ordered list, and then requests the list. This results in the object taking a step in a random direction, but not in a direction that brings it too close to the walls of the corridor (because in this case the relevant event is blocked).

If the set of directions is chosen correctly (the average direction of the set is always in the general direction of the corridor) the object succeeds in getting from one side of the corridor to the other.

An example of this animation can be viewed here:
https://www.dropbox.com/s/mqd239tu02xjbau/Sec5Sub3.mp4.

6 Flock Movements

Boids is an algorithm that simulates the flocking behavior of birds; see [23]. The basic algorithm consists of three rules that each bird follows:

1. Separation- maintain a small distance from other birds.
2. Alignment- try to match the velocity with the average velocity of the flock.
3. Cohesion- fly towards the center of mass of the flock.

Other behaviors can be added, such as flying away from the center of mass of the flock when there is a threat, maintaining a certain minimum and maximum speed, and keeping away from walls and other objects. In this example, the birds fly in a flock as long as there is no threat. When a threat occurs (simulated in our example by a mouse left-click) the birds fly away from the flock. When they stop flying in a united flock (the mouse is right-clicked) the birds continue flying, keeping away from each other and other objects, and they do not follow the alignment and cohesion rules.

Since this algorithm is based on the birds' individual behaviors, it can be easily implemented using behavioral programming. We set things up so that every bird in the flock has a motion b-thread for each rule. Control b-threads are used to block events requested by motion b-threads that represent rules that are not relevant to the state of the bird or flock.

Each bird has the following motion b-threads:

- **Boid-** Every time tick, this b-thread updates the coordinates according to the velocity and the time passed. This is the main motion b-thread.
- **MatchSpeed-** Every time tick, this b-thread requests the event `matchSpeed` and updates the velocity so that it is closer to the average velocity of the rest of the flock.
- **FlyTowardsCenterOfMass-** Every time tick, this b-thread requests the event `flyTowardsCenterOfMass`, and moves the bird towards the center of the flock.
- **FlyAwayFromCenterOfMass-** Every time tick, this b-thread requests the event `flyAwayFromCenterOfMass` and updates the velocity so that the bird flies away from the center of the flock.
- **KeepSpeed-** Every time tick, this b-thread updates the velocity of the bird to keep it between a given minimum and maximum.
- For every wall K **SoftBounceFromKWall-** Every time tick, this b-thread requests the event `softBounceFromKWall` and updates the velocity of the bird to make it move away from the wall.
- For every wall K **HardBounceFromKWall-** Every time tick, this b-thread requests the event `hardBounceFromKWall` and reverses the velocity of the bird to make it move away from the wall.
- For every other bird i in the flock **KeepAway-** Every time tick, this b-thread requests the event `keepAway[i]` and moves the bird away from bird i.

We have the following control b-threads:

- **MouseReleased-** When the mouse is released from its left-click, the b-thread is created. It waits for the event `mousePressed` and blocks `flyAwayFromCenterOfMass`. This makes the birds fly as a flock with all the relevant behaviors.

- **Scared-** When the mouse is left-clicked, the b-thread is created. It waits for the event `mouseReleased` and blocks `matchSpeed` and `flyTwardsCenterOfMass`. This is so that the birds fly away from each other as fast as possible, to avoid the threat.
- **NotCooperative-** When the mouse is right-clicked, the b-thread is created. It waits for the event `mouseReleased` and blocks `matchSpeed`, `flyTwardsCenterOfMass` and `flyAwayFromCenterOfMass`. This way the birds continue to move, but not as a united flock.

Every bird in the flock has the following control b-threads:

- For every wall K **CheckKWall-** Every time tick, this b-thread checks where the bird is with respect to the wall, and then blocks `hardBounceFromKWall` and `softBounceFromKWall` accordingly.
- For every pair of birds in the flock **CheckCollision-** Every time tick, this b-thread blocks the `keepAway` event of both birds if they are not close to each other. This is so they will not fly away from each other when there is no need to do so.

The b-threads above implement the Boid algorithm using BP. It is an example of how animation with BP integrates defining local rules between various objects that have synthetic vision with BP, to further simplify the creation of complex computerized animations.

Examples of this animation can be viewed here:
https://www.dropbox.com/s/g6zu5n34psxaoi3/flock.mp4, and here:
https://www.dropbox.com/s/yt7lhixhukuyu35/flock%205%20X2.avi

The boid animation was based on open source code that can be downloaded here: http://ultrastudio.org/en/Project:Boids.

7 Future Work

In the billiard and flock examples, each object had a number of behaviors. Since each of these was turned into a b-thread, there are many context switches between b-threads before the new coordinates of an object can be calculated. It takes a while to execute these context switches, because the system needs to check that the next event to be triggered is not blocked by any b-thread, so that it has to be compared to all the blocked events in the program.

In these two examples there is a relatively large number of objects. This can create a problem if the time between each clock tick is too short. When this occurs, an object with high priority can execute two or more moves, while an object with a lower priority does not move at all. This happens when the `endSleep` event of the higher priority object is requested before the `endSleep` event of the lower priority object is triggered, which results in the `endSleep` event of the higher priority object being triggered instead of the `endSleep` event of the lower priority object. When this happens, it can be seen on-screen; some of the objects move, while others do not. To solve this problem, the context switch between b-threads should be optimized.

In addition to optimizing the algorithm for the context switch between b-threads, further work should be done on behavioral programming with multiple time scales [20]. Our work on programming animation using BP enables using a different clock, and hence a different time scale, for each object. Although objects then move independently of each other, they still share common variables and events. More work can be done on rendering the behaviors of these objects truly independent, while still synchronizing their execution.

Our work simplifies programming animation, but is still far from becoming as simple as we would like. Research could be done on analyzing how animation is described informally in layman's terms, and then using the results to suggest formal programming language primitives to enhance the BP paradigm with means for specifying animation. Also, the physical calculations in this article are carried out in conventional code, and it would be beneficial for users to have a system that incorporates a feature that enables making these calculations directly from a mathematical formula. In addition, functionality should be added to BP, to give the user a better illusion that motion b-threads are actually continuous. This will make programming animation more user-friendly to non-programmers.

Additional work can also be done on more complex animation examples. What comes to mind are compound objects that have many moving parts, such as worm or a human. Another example is of an object with dynamic boundaries, such as a stress ball. Other complex animations would involve the merging and splitting of objects; cells, for example.

Acknowledgements. Part of this research was supported by the I- CORE program of the Israel Planning and Budgeting Committee and the Israel Science Foundation.

References

1. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001)
2. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, New York (2003)
3. Harel, D., Maoz, S., Szekely, S., Barkan, D.: PlayGo: towards a comprehensive tool for scenario based programming. In: Proceedings of the IEEE/ACM 25th International Conference on Automated Software Engineering (ASE 2010), Antwerp, Belgium, pp. 359–360 (2010)
4. Harel, D., Marron, A., Weiss, G.: Programming coordinated behavior in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 250–274. Springer, Heidelberg (2010)
5. Harel, D., Kantor, A., Katz, G.: Relaxing synchronization constraints in behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR-19 2013. LNCS, vol. 8312, pp. 355–372. Springer, Heidelberg (2013)
6. Wiener, G., Weiss, G., Marron, A.: Coordinating and visualizing independent behaviors in erlang. In: Fritchie, S.L., Sagonas, K.F. (eds.) Erlang Workshop, pp. 13–22. ACM (2010)

7. Marron, A., Weiss, G., Wiener, G.: A decentralized approach for programming interactive applications with javascript and blockly. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE! 2012, pp. 59–70. ACM, New York (2012)
8. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**(7), 90–100 (2012)
9. Devillers, F., Donikian, S.: A scenario language to orchestrate virtual world evolution. In: SCA 2003: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Aire-la-Ville, Switzerland, Switzerland, pp. 265–275. Eurographics Association (2003)
10. Atir, Y., Harel, D.: Using LSCs for scenario authoring in tactical simulators. In: Proceedings of Summer Computer Simulation Conference (SCSC 2007), pp. 437–442 (2007)
11. Yu, Q., Terzopoulos, D.: A decision network framework for the behavioral animation of virtual humans. In: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 119–128. Eurographics Association (2007)
12. Haumann, D.R., Parent, R.E.: The behavioral test-bed: obtaining complex behavior from simple rules. Vis. Comput. **4**(6), 332–347 (1988)
13. Renault, O., Magnenat-Thalmann, N., Cui, M., Thalmann, D.: A vision-based approach to behavioral animation (1990)
14. Noser, H., Thalmann, D.: Sensor-based synthetic actors in a tennis game simulation. Vis. Comput. **14**(4), 193–205 (1998)
15. Peters, C., O'Sullivan, C.: Bottom-up visual attention for virtual human animation. In: 16th International Conference on Computer Animation and Social Agents, pp. 111–117. IEEE (2003)
16. Ondřej, J., Pettré, J., Olivier, A.H., Donikian, S.: A synthetic-vision based steering approach for crowd simulation. ACM Trans. Graph. (TOG) **29**, 123 (2010). ACM
17. Goldenstein, S., Karavelas, M., Metaxas, D., Guibas, L., Aaron, E., Goswami, A.: Scalable nonlinear dynamical systems for agent steering and crowd simulation. Comput. Graph. **25**(6), 983–998 (2001)
18. Treuille, A., Cooper, S., Popović, Z.: Continuum crowds. ACM Trans. Graph. (TOG) **25**, 1160–1168 (2006). ACM
19. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems, pp. 209–229 (1992)
20. Harel, D., Marron, A., Wiener, G., Weiss, G.: Behavioral programming, decentralized control, and multiple time scales. In: Lopes, C.V. (ed.) SPLASH Workshops, pp. 171–182. ACM (2011)
21. Shimony, B., Nikolaidis, I., Gburzynski, P., Stroulia, E.: On coordination tools in the picos tuples system. In: Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications, SESENA 2011, pp. 19–24. ACM, New York (2011)
22. Kugler, H., Plock, C., Roberts, A.: Synthesizing biological theories. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 579–584. Springer, Heidelberg (2011)
23. Reynolds, C.W.: Flocks, herds and schools: a distributed behavioral model. In: Stone, M.C. (ed.) SIGGRAPH, pp. 25–34. ACM (1987)