A dynamically changing spline curve for the motion of a game character

Dag Nylund, Ole E. Flaten

Abstract

In computer games a frequent gameplay mode consists of a non-player character (NPC) which patrols around to defend certain items. The player tries to collect these items for her inventory. The NPC's role is to make this challenging (in an entertaining way) for the player. The NPC's patrol path is often implemented by using the A* algorithm. In this paper we present an algorithm which uses B-splines and basic learning concepts to construct the NPC's path, and its motion and behavior in general. We look at some learning strategies for the NPC and use gamedesign concepts to construct the spline curves. The NPC's path will be a dynamically changing spline curve based on gameplay principles and the NPC's knowledge.

1 Introduction

A computer game has one or more players and non-player characters (**NPC**s), rules and a winning condition, and the purpose of a game is usually entertainment. According to ([1], p. 9 and 41) a **gameplay** consists of *challenges and actions and is the primary source of entertainment in a game*. A game can be divided in levels and scenes with static and dynamic objects. The customisation of a scene (art) and the behavior of NPCs are factors that contribute to a good gameplay.

In computer games a frequent gameplay mode ([1], p. 40-41) contains a NPC which patrols around to defend certain items. Figure 1 illustrates this situation. There are three **item**s represented by stars which a NPC should try to defend. The NPC will patrol along the **path** which is a cubic **spline curve** with five **control point**s - the two endpoints marked by squares and the three items' positions.

The spline curve interpolates the endpoints and approximates the items to be protected. In our algorithm we change the patrolling path each time the NPC arrives at an endpoint. When the player collects a certain item there is no longer any reason for the NPC to defend that item. The item's position is removed from the set of control points from which a new path is constructed.

We implement some **basic learning** for the NPC. We discuss the problem to solve from a gameplay perspective and explain how we use basic learning and spline methods to construct the paths. First we will briefly describe how paths are generated in commercial game engines.

Pathfinding algorithms in computer games often use a grid, either square blocks through the scene, or a navmesh - a mesh that contains the walkable area of the scene. The blocks

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

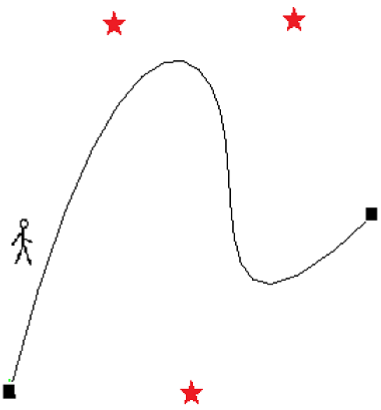

Figure 1: A NPC patrols on its path to defend the items represented by stars, to make it challenging for the player to collect the items. The interpolating endpoints are marked with squares. The path is a cubic spline curve.

or the polygons of these structures can be used as nodes in a graph. Often the A* algorithm or Dijkstra's algorithm is used to find the shortest path between selected nodes in this graph ([5], [6]).

The commercial game engines Unity and Unreal Engine 4 have tools to generate navmeshes and shortest paths. When these tools are applied the NPC will follow a path of straight lines, and walk through or very close to the specified points (figure 2). There is no inherent way to force the NPC to follow a smooth path that only passes by the given points. The engines have to use additional algorithms to achieve this. We solve it by the properties of spline curves.

The paths we construct are not shortest paths, but natural paths between the chosen points the NPC has to guard. Our algorithm does not guarantee a path free from obstacles, but obstacles will be learned. The features of the proposed solution are

- The patrol route is changed each time the NPC arrives at an end node. Thereby we can omit that the player learn the NPC's route.
- The NPC use the tangent vector of the spline curve at its position as the forward

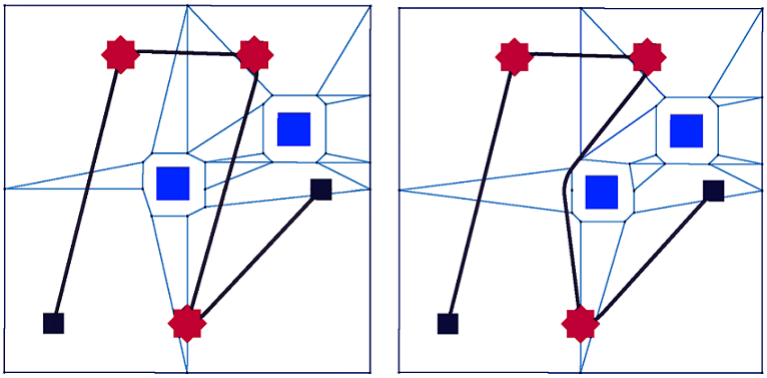

Figure 2: Patrolling paths with endpoints and items as in figure 1, built by Unity's navmesh and pathfinding algorithms. The blue squares show obstacles and the blue lines represent navmeshes. To the right we see the algorithm avoid an obstacle.

vector. It will change direction in a natural, smooth way throughout the path. This is not always the case with other algorithms.

- The spline curves will in some ways be constructed based on the NPC's knowledge learned by patrolling.

In the rest of this paper we use the words path and spline curve interchangeably.

We expect from the NPC that it actually learns something by patrolling, for example that it should sometimes change its path. We let the items be equally ranked, so there is no preferable order of the control points.

We look at four kinds of learning according to the events *endpoint arrived, item collected, player detected* and *obstacle detected*. We construct quadratic and cubic spline curves in 2D. A spline curve is determined by its:

1. **degree** d
2. n control points $\mathcal{P} = \{p_i\}$ which are points in 2D or 3D. Connect the points $(p_i,\ p_{i+1})_{i=0}^{n-2}$ by straight lines to construct the control polygon.
3. n B-spline basis functions of degree d - to be multiplied by the control points. (1)
4. A **knot vector** with $n+d+1$ knots $\mathbf{t} = \{t_i\}_{i=0}^{n+d}$.

Note that the degree d is something different than the dimension of the spline curve's space.

2 Using a finite state machine

We implement some basic learning for the NPC and in this context different **states** are used. We use a finite state machine (FSM) as explained in ([2], p.41). We list the states used here and give a short description. We use the concepts **event, notification** and **transition** in this context, though the notifications are sent from the NPC to itself. The reason for this is that not all transitions can be done immediately. In the state patrol the NPC is also learning, but the knowledge achieved can not always be used at once. Therefore a notification is sent, and the corresponding transition is performed in the learn state. We use state names as suggested in ([2], p.43) and some new state names.

patrol state is the state of patrolling along paths to protect items, and the state in which the NPC will be most of the time. The NPC knows the spline curve, the elapsed time and the knot interval. In patrol state the NPC

- detects the knot interval i and computes its position. If the NPC detects that an item is taken a notification is sent.
- observes if the player is within a certain distance or angle of view. A notification is sent if this happens.
- detects if there is an obstacle within a short distance. If this happens, a notification is sent.

learn state is the state where all notifications are taken care of. This state normally follows after patrol state and occurs when the NPC arrives at an endpoint.

chase state is entered if the NPC discovers the player during patrolling. In this state a secondary curve is used.

	state	**event**	**notification**	**transition**
1	patrol	path done	ENDPOINT_ARRIVED	state=learn
2	patrol	item taken	ITEM_TAKEN	state=learn
3	patrol	obstacle detected	OBSTACLE_DETECTED	state=learn
4	patrol	player detected	PLAYER_DETECTED	state=learn
	state	**event**	**transition**	**transition**
5	learn	item taken	remove item, build new path	state=patrol
6	learn	endpoint arrived	build new path	state=patrol
7	learn	obstacle detected	build obstacle path	state=patrol
8	learn	player detected	build chase path	state=chase

The transitions in line 1-2 are done at the path's end. The transitions in line 3-4 are done immediately.

3 Using spline curves

The characteristics of a spline curve is given in (1). All the paths constructed here are either quadratic or cubic spline curves which interpolate the endpoints and approximate items. See figure 7 for a comparision between a quadratic and a cubic curve. We briefly summarize some of the properties of the spline curves applied.

A spline curve is constructed from several B-spline basis functions, and the knots are the parameter values where the basis functions join. A knot vector is often written as $\mathbf{t} = \{t_0, t_1, \ldots, t_{n+d}\}$ where $t_i \leq t_{i+1}$. Multiple knots are allowed but at least for two knots this inequality is strict. A B-spline basis function of degree d is positive on a knot interval $(t_i, \ldots, t_{i+d+1})$ and zero elsewhere. That means we need d+2 knots for each B-spline basis function of degree d. When a spline curve has n control points we need $n+d+1$ knots since the curve is constructed by blending control points with B-spline basis functions:

$$\mathbf{f}(t) = \Sigma_{i=0}^{n-1}\, c_i \cdot B_{i,d}(t) \tag{2}$$

Here the function value $\mathbf{f}$ returned can be a vector in $\mathbf{R}^2$ or $\mathbf{R}^3$ or just a real number. So the spline curves we use are defined by the degree d, n control points c_0, $c_1, \ldots c_{n-1}$ and knots $t_0, \ldots, t_{n+d}$. The spline curves will be evaluated using Cox-deBoor algorithm ([3], ch. X). We now list some of the properties of spline curves which we apply:

- Given $n \geq d+1$ control points ($n-2$ items) it is necessary to use $n+d+1$ knots to obtain a cubic C^2 spline curve or a quadratic C^1 spline curve which interpolates the endpoints. Use $d+1$ equal knot values in each endpoint. If $n > d+1$ add one interior knot for each extra control point to obtain C^2 continuity for a cubic spline curve or C^1 continuity for a quadratic spline curve.

- Examples of knot vectors for spline curves with 5 control points and 5 basis functions:

  - Construct a C^2 cubic spline curve which has $n = 5$ control points and interpolates the endpoints as in figure 1 by selecting the $n+d+1 = 9$ knots $\{0,0,0,0,\frac{1}{2},1,1,1,1\}$.
  - Construct a C^1 quadratic spline curve which has n=5 control points and interpolates the endpoints by selecting the $n+d+1 = 8$ knots $\{0,0,0,\frac{1}{3},\frac{2}{3},1,1,1\}$.

- Remove the middle interior knot each time a control point is removed (an item is collected) to preserve the continuity of the curve. In general, use equally spaced interior knots.

Consider now that the NPC detects that the item at control point c_i is taken. If c_i is deleted and a new spline curve is calculated immediately, the NPC's position will no longer be on the curve and there will be a discontinuity in its motion in the next frame. Therefore the path should not be reconstructed until the NPC arrives at an endpoint. The endpoints are the only points which all paths interpolate and thus the only points which are guaranteed to be included in the new path.

The example spline curves representing the patrolling path shown in this paper are constructed from the control points

$$c_0(0,0),\ c_1(\tfrac{1}{2},2),\ c_2(1,0),\ c_3(\tfrac{3}{2},2) \text{ and } c_4(2,1).$$

The notation 01234 is used to denote a path with these control points in initial order. Curves constructed by different permutation of the initial order are given names indicating the permutation (02134, etc.).

4 The algorithms

We now look at how to solve the different tasks discussed above. In addition we discuss chasing the player and obstacle detection later in the section. Some of the details in the algorithms are omitted.

In this section we suppose we have a spline function as described in (1) and a function

```
Vec2 deBoor(float x)
{ //return curve position calculated by deBoor's algorithm }
```

The finite state machine

The finite state machine algorithm tells about states used:

```
while (game running)
    if (npc_state == PATROL) patrol()
    if (npc_state == CHASE) chase()
    if (npc_state == LEARN) learn()
```

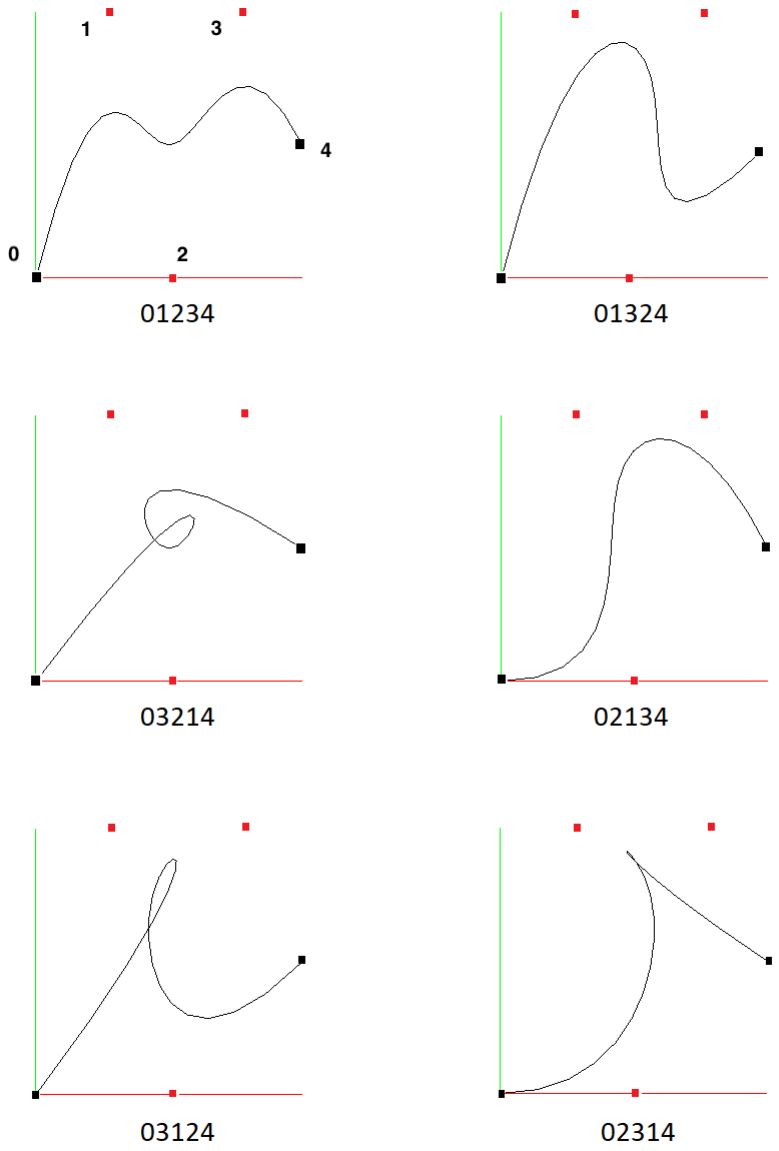

Figure 3: Curves with permutations of the control points. The upper left curve shows the initial control point order.

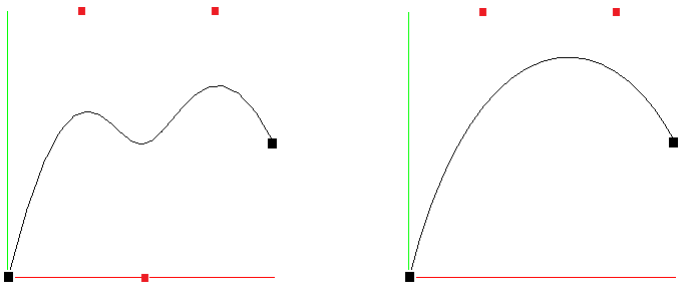

Figure 4: Cubic curve before and after item 1 (control point 2) is collected.

The patrol state

Patrolling is discussed earlier in this paper, so we just give the algorithm here:

```
void patrol() {
    position = deBoor(t)
    draw NPC
    if (endpoint arrived)
        notify(endpoint_arrived)
        npc_state = LEARN
    if (item collected)
        notify(control_point_index)
    if (player detected)
        notify(player_position)
        npc_state = LEARN
    else if (obstacle detected)
        notify(obstacle_position, control_point_index)
        npc_state = LEARN
    else
        previous_position = position
}
```

The learn state

The following algorithm shows the main idea in this paper. At each endpoint arrival points and knots can be removed and a new path is constructed. The continuity of the curve is preserved.

```
void learn() {
    if (item collected)
        remove controlpoint and one (the middle) internal knot
    if (player_detected)
        build chase path by Chasing path algorithm
        npc_state = CHASE
    if (obstacle_detected)
        mark edge unavailable
        build obstacle path by Obstacle path algorithm
        npc_state = CHASE
    if (endpoint arrived)
```

```
        build new path
        npc_state = PATROL
    if (all items collected)
        stop
}
```

Here the chase state follows after build an obstacle path. The patrol state can also be used here.

The chase state

If the NPC discovers the player, a new path is built immediately so the NPC can start chasing the player. We expect that the player's position is known to the NPC. Our approach to the chase state is to build a temporary path from the NPC's current position towards the player's position and then towards an endpoint. The chase state will resemble the patrol state. The only difference is that the NPC follows another path, so we don't the algorithm for the chase state.

The temporary path will consist of four control points stored in an own array **cp** and it will have its own knot vector **u**. $\text{NPC}_{current}$, $\text{Player}_{current}$ and $\text{NPC}_{previous}$ denote positions.

We want a smooth change in direction for the NPC when it changes path. The tangent vector to the curve is obtained by differentiating the spline function. The derivative of a B-spline of degree d is given by ([4], equation 7.12) and can be computed by using the deBoor algorithm on two spline functions of degree $d-1$. Here we simply evaluate the expression

$$\mathbf{v} = \frac{\text{NPC}_{current} - \text{NPC}_{previous}}{t_{current} - t_{previous}} \tag{3}$$

for the derivative and construct the chasing path as follows.

Chasing path algorithm

$cp_0 = NPC_{current}$
$cp_1 = NPC_{current} + \alpha \cdot \mathbf{v},\ \alpha \in \mathbf{R}$
$cp_2 = Player_{current}$
$cp_3 = c_{n-1}$
// use degree 3
$u_i = t_0;\ i = 0 \ldots 3$
$u_i = t_{n+d};\ i = 4 \ldots 7$

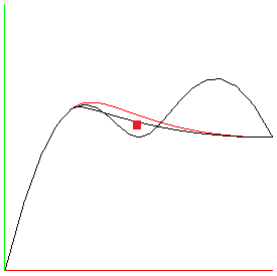

Figure 5: Chasing paths for the NPC when patrolling path 01234. The player is marked with a square. The α values in chasing path algorithm line 2 in this example are $\alpha = 0.2$ (black) and $\alpha = 0.5$ (red).

The chasing paths in figure 5 are cubic spline curves. A quadratic curve can also be constructed by a small change of the knot vector.

Obstacle detection

We suppose that the NPC is able to detect obstacles shortly before a collision would happen. The NPC should learn not to use a new path which contains the current knot interval and it should try to get around the obstacle (though in an early level of the game returning to an endpoint will be sufficient). A path around the obstacle can be constructed as follows: Given vectors

$$\mathbf{v} = (a,b),\ \mathbf{w} = (1,w_y) \text{ and } (\mathbf{v},\ \mathbf{w}) = 0 \Leftrightarrow \mathbf{w} = (1,\ -\frac{a}{b}) \tag{4}$$

With the tangent vector $\mathbf{v}$ from (3), this vector $\mathbf{w}$ and appropriate scaling parameters α, β, a path for trying to get around the obstacle can be constructed in the following way:

Obstacle path algorithm

Note that this algorithm differs from Chasing path algorithm only in line 3. The knot vectors in both algorithms are chosen such that the *endpoint arrived* test in patrol state can be used.

$cp_0 = NPC_{current}$
$cp_1 = NPC_{current} + \alpha \cdot \mathbf{v}$
$cp_2 = cp_1 + \beta \cdot \mathbf{w}$
$cp_3 = c_{n-1}$
// use degree 3
$u_i = t_0;\ i = 0 \ldots 3$
$u_i = t_{n+d};\ i = 4 \ldots 7$

Also here a quadratic curve can be chosen.

Figure 6 shows an obstacle path generated from position (0.5, 1.22) with parameters $\alpha = 5.0$ and $\beta = -0.2$.

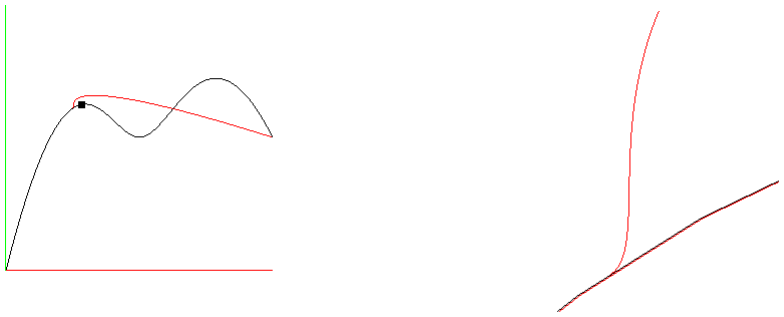

Figure 6: Obstacle path example. The curve to the right are zoomed in to show the C^1 continuity where the curves split.

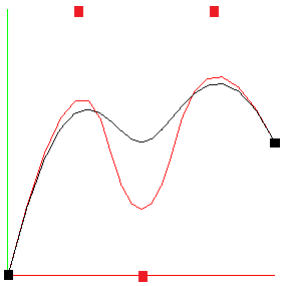

Figure 7: Cubic spline curve on the knotvector $\{0,0,0,0,1,2,2,2,2\}$ and quadratic spline curve on the knotvector $\{0,0,0,\frac{2}{3},\frac{4}{3},2,2,2\}$ for the original control point sequence.

5 Conclusion

We have described how B-spline curves can be applied to construct a dynamically changing path for a NPC. The main idea is to permute the control points during patrolling to change the path and thereby contribute to improved gameplay. Using spline curves for this purpose will produce different paths than using the similar tools in commercial game engines (see section 1 and figure 2). Whether the gameplay is improved or not has to be tested in each specific game.

In the chasing path algorithm the player's position is detected during patrolling. Once the chase state is entered the NPC follows the chasing path without rechecking the player's position. This solution can be sufficient in an easy level. A more challenging chasing path can be achieved by checking the player's position and build a new chasing path if player movement is detected.

The obstacle path algorithm is experimental. We have shown that a path around an obstacle can be constructed with appropriate control points, but not how these points should be selected in general (through choice of parameters α and β in the algorithm).

In most examples we have used cubic spline curves. The curves constructed are paths which will normally not be rendered, so for our purpose we do not need the smoothness of a cubic curve. Figure 7 suggests the use of quadratic curves.

References

[1] E.Adams *Fundamentals of Game Design Third Edition*, New Riders 2014

[2] J.B.Ahlquist, jr., J.Novak *Game Artificial Intelligence*, Thomson 2008

[3] C.deBoor *A Practical Guide to Splines*, Springer-Verlag 1978

[4] R.Goldman *Pyramid Algorithms*, Morgan Kaufman Elsevier 2003

[5] Y.Langsam, M.J.Augenstein, A.M.Tenenbaum *Data Structures Using C and C++*, Prentice-Hall 1996

[6] M.A.Weiss *Data Structures and Algorithm Analysis i C++ Fourth Edition*, Pearson 2014