BDCFF Objects: Index

This is a list of all the currently defined BDCFF Object Types, grouped by the implementation from which they originally come.

Use these specifications to implement the objects in your Boulder Dash implementation. If any part of any specification seems in any way ambiguous, please email me and I shall see about resolving the ambiguity.

Each object spec includes some properties at the top of their web page. You can check out the meaning of the properties in the BDCFF Object Properties page.

From *Boulder Dash* (by Peter Liepa)

BDCFF Object 0000: Boulder

Object number: $0000
Game class: Boulder Dash (by Peter Liepa)
Object name: Boulder

In this document:

Properties

Animate: yes
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: yes

Attributes

Attribute format: %00000000 0000000a

a: Flag indicating whether the Boulder is currently considered to be "falling" or "stationary". The flag is set (1) when falling, clear (0) when stationary. It is recommended that all boulders begin life as stationary boulders.

Graphics

This GIF shows the graphic of a boulder from the C64 implementation of Boulder Dash (hence the graphic is 8 double-width pixels wide and 16 pixels high). The boulder does not have an animation sequence: it looks the same all the time.

Interactions with other objects

The boulder interacts with the following objects:

- Objects it can roll off: any object which is defined to be "rounded" (brick wall, boulder, diamond). However, in the special cases of rolling off boulders and diamonds, the boulder or diamond which it is rolling off must be stationary.
- Objects it can cause to explode: any object which is defined to be "explosive" (firefly, butterfly, Rockford).
- Magic wall

Specification

The boulder, like the diamond, is an object which falls, rolls off some other objects, and can explode some other objects when it hits them.

Falling

- A stationary boulder which is discovered to have space underneath it changes state into a falling boulder (with an appropriate sound played) and moves down one position.
- A falling boulder which has space underneath it continues to fall another position.
- A falling boulder discovered to *not* have space underneath it will have different effects depending on the object below:
    - Cause explosion: if the object below is explosive (firefly, amoeba, Rockford), the boulder will cause the object below to explode
    - Magic wall: see below
    - Stop falling: for any other object below, an appropriate sound is played as of a boulder hitting an object. In addition, a check is made to see whether the boulder can roll (see below); if so, the boulder is moved to its new position and is still considered falling, otherwise the boulder remains in its current position but changes state into a stationary boulder.

Falling through magic wall

If a falling boulder hits a magic wall, then:

1. A sound is played: the sound is as of a *diamond* hitting something. This sound is played regardless of what happens next.
2. If the magic wall was Dormant (a global attribute), the magic wall is now considered to be On.
3. If the magic wall is now On then
    - if there is space in the position below the magic wall then the boulder morphs into a falling diamond and moves down *two* positions, to be below the magic wall
    - otherwise the boulder simply disappears

A stationary boulder sitting on top of a magic wall does not activate the magic wall or move in any way (not even rolling off it).

Rolling

Note that both stationary and falling boulders can roll.

If a boulder is discovered to have a stationary, rounded object (stationary boulder, stationary diamond, brick wall) below it, then the boulder will attempt to roll off the object below. Note that falling boulders and diamonds can roll off things too; they don't have to come to a halt first.

In order for a boulder or diamond to roll, not only must the object below be a brick wall, stationary boulder or stationary diamond, but the objects to the left and diagonally left/down (or right and diagonally right/down) must both be space. Preference is given to rolling to the left over rolling to the right. If these criteria are satisfied, the boulder or diamond is moved one space immediately to the side (not diagonally down) and is changes state to be falling (if it wasn't already).

If the boulder is not able to roll, it remains where it is, and changes state into a stationary boulder (if it wasn't already).

Causing explosions

If a falling boulder hits an explosive object (firefly, butterfly, Rockford), then that object explodes, consuming the boulder in the explosion. The boulder itself does not explode.

General Algorithm

```
procedure ScanStationaryBoulder(in positionType boulderPosition)
# Local variables
 positionType NewPosition;
 objectType theObjectBelow;

# If the boulder can fall, move it down and mark it as falling.
 NewPosition := GetRelativePosition(boulderPosition, down1);
 theObjectBelow := GetObjectAtPosition(NewPosition);
 if (theObjectBelow == objSpace) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition);
 RequestSound(boulderSound); # yes, even when it starts falling. This
applies to diamonds too (requests diamondSound).
 else

# Failing that, see if the boulder can roll
 if (CanRollOff(theObjectBelow)) then

# Try rolling left
 NewPosition := GetRelativePosition(boulderPosition, left1);
 if ((GetObjectAtPosition(NewPosition) == objSpace) and
(GetObjectAtPosition(GetRelativePosition(boulderPosition, down1left)) ==
objSpace)) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition);
 else

# Try rolling right
 NewPosition := GetRelativePosition(boulderPosition, right1);
 if ((GetObjectAtPosition(NewPosition) == objSpace) and
(GetObjectAtPosition(GetRelativePosition(boulderPosition, down1right)) ==
objSpace)) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition);
 endif
 endif
 endif
 endif
endprocedure

##
```

```
procedure ScanFallingBoulder(in positionType boulderPosition;
 in/out magicWallStatusType magicWallStatus)
# Local variables
 positionType NewPosition;
 objectType theObjectBelow;

# If the boulder can continue to fall, move it down.
 NewPosition := GetRelativePosition(boulderPosition, down1);
 theObjectBelow := GetObjectAtPosition(NewPosition);
 if (theObjectBelow == objSpace) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition); # ie old position

# If the object below is a magic wall, we activate it (if it's off), and
# morph into a diamond two spaces below if it's now active. If the wall
# is expired, we just disappear (with a sound still though).
 elsif (theObjectBelow == objMagicWall) then
 if (magicWallStatus == kMagicWallOff) then
 magicWallStatus := kMagicWallOn);
 endif
 if (magicWallStatus == kMagicWallOn) then
 NewPosition := GetRelativePosition(boulderPositon, down2);
 if (GetObjectAtPosition(NewPosition) == objSpace) then
 PlaceObject(objDiamond, attribFalling, NewPosition);
 endif
 endif
 PlaceObject(objSpace, attribNone, boulderPosition);
 RequestSound(diamondSound); # note: Diamond sound
 endif

# Failing that, we've hit something, so we play a sound and see if we can
roll.
 else
 RequestSound(boulderSound);
 if (CanRollOff(theObjectBelow)) then

# Try rolling left
 NewPosition := GetRelativePosition(boulderPosition, left1);
 if ((GetObjectAtPosition(NewPosition) == objSpace) and
(GetObjectAtPosition(GetRelativePosition(boulderPosition, down1left)) ==
objSpace)) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition);
 else

# Try rolling right
 NewPosition := GetRelativePosition(boulderPosition, right1);
 if ((GetObjectAtPosition(NewPosition) == objSpace) and
(GetObjectAtPosition(GetRelativePosition(boulderPosition, down1right)) ==
objSpace)) then
 PlaceObject(objBoulder, attribFalling, NewPosition);
 PlaceObject(objSpace, attribNone, boulderPosition);

# The boulder is sitting on an object which it could roll off, but it can't
# roll, so it comes to a stop.
 else
 PlaceObject(objBoulder, attribStationary, boulderPosition);
 endif
 endif

# Failing all that, we see whether we've hit something explosive
```

```
 elsif (ImpactExplosive(theObjectBelow) then
 Explode(NewPosition, GetExplosionType(theObjectBelow));

# And lastly, failing everything, the boulder comes to a stop.
 else
 PlaceObject(objBoulder, attribStationary, boulderPosition);
 endif
 endif
endprocedure

##

function CanRollOff(in objectType anObjectBelow):Boolean
# If the specified object is one which a boulder or diamond can roll off,
# return true otherwise return false.

# First of all, only objects which have the property of being "rounded" are
# are ones which things can roll off. Secondly, if the object is a boulder
# or diamond, the boulder or diamond must be stationary, not falling.

# We're going to assume that GetObjectProperty() automatically returns
"true"
# for objBoulderStationary, objDiamondStationary, objBrickWall, and returns
"false"
# for everything else (including objBoulderFalling and objDiamondFalling).

 return (GetObjectProperty(anObjectBelow, propertyRounded));
endfunction

##

function ImpactExplosive(in objectType anObject):Boolean
# If the specified object has the property of being something that can
# explode, return true otherwise return false.
# ImpactExplosive objects are: Rockford, Firefly, Butterfly.
 return (GetObjectProperty(anObject, propertyImpactExplosive)); #
true/false
endfunction

##

function GetExplosionType(in objectType anObject):explosionType;
# Assuming that the specified object is in fact explosive, returns the type
# of explosion (explodeToSpace or explodeToDiamonds)
# Explosive objects are: Rockford, Firefly, Butterfly.

 ASSERT (Explosive(anObjectBelow));

 return (GetObjectProperty(anObject, propertyExplosionType));
endfunction

##
```

BDCFF Object 0001: Diamond

Object number: $0001
Game class: Boulder Dash (by Peter Liepa)
Object name: Diamond (or Jewel?)

In this document:

- Properties
- Attributes
- Graphics
- Specification

Properties

Animate: yes
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: yes

Attributes

Attribute format: %00000000 0000000a

a: Flag indicating whether the diamond is currently considered to be "falling" or "stationary". The flag is set (1) when falling, clear (0) when stationary. It is recommended that all diamonds begin life as stationary diamonds.

Graphics

This GIF shows the animation sequence of a diamond from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

Specification

The diamond is almost identical to the boulder. Basically, you can take the boulder's algorithm and replace all instances of "boulder" with "diamond", and in the case of falling through the magic wall, a diamond morphs into a boulder in the same way that boulders morph into diamonds.

Please refer to the boulder specification for full details.

BDCFF Object 0002: Magic wall

Object number: $0002
Game class: Boulder Dash (by Peter Liepa)
Object name: Magic wall (or Enchanted wall?)

In this document:

Properties

Animate: no
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type. (There is a global attribute, being the state of the wall: **dormant**, **on** or **expired**, but that doesn't apply to individual pieces of magic wall.)

Graphics

When **dormant** or **expired**, the magic wall looks like ordinary brick wall:

When **on**, the magic wall animates in a 4-stage animation sequence:

This GIFs show the graphics from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

Interactions with other objects

Magic wall is inanimate. It doesn't actually do anything by itself, and therefore doesn't interact with any other objects.

Instead, other objects (boulder, diamond) interact with magic wall.

Specification

Magic wall starts life **dormant**, and visually looks just like ordinary brick wall, until a boulder or diamond falls on it. Then all magic wall in the cave gets **turned on** for a time: it sparkles in a distinctive way, it makes a continous tinkling sound, and in this time any boulder or diamond that hits the wall gets changed into a diamond or boulder respectively, and falls through, making an appropriate diamond sound or boulder sound respectively. After a certain time has elapsed (the magic wall milling time), the magic wall **expires**; it goes back to looking like ordinary brick wall, no longer makes any sound, and any boulder or diamond that fall on the wall just disappear (but still make the same sound as appropriate).

States of magic wall

1. **Dormant**: Looks just like ordinary brick wall, until a boulder or diamond activates it.
2. **On**: Sparkles, makes sound, and changes boulders and diamonds into diamonds and boulders, until the Magic Wall Milling Time runs out.
3. **Expired**: Looks just like ordinary brick wall; any boulders and diamonds hitting it just disappear but still with a sound.

Objects move two squares in one frame

Note that when a boulder or diamond hits a magic wall (whether **on** or not), the object effectively moves two squares in one frame: one moment it's above the wall, the next moment it's two squares down, below the wall; it's never "inside" the wall. Note also that if there is any object underneath the wall at the point where the boulder or diamond falls through, the boulder or diamond is lost (which is sometimes a helpful way of getting rid of excess boulders).

Stationary objects don't fall through

Note that a stationary boulder or diamond sits on a magic wall without falling through or turning it on. Note also that magic wall isn't "rounded"; boulders and diamonds do not roll off magic wall like they do brick wall.

General Algorithm

See the boulder specification for how boulders (and diamonds) interact with magic wall.

BDCFF Object 0003: Brick wall

Object number: $0003
Game class: Boulder Dash (by Peter Liepa)
Object name: Brick wall

In this document:

Properties

Animate: no
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: yes

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics

This GIF shows the graphic of brick wall from the C64 implementation of Boulder Dash (hence the graphic is 8 double-width pixels wide and 16 pixels high). The object does not have an animation sequence: it looks the same all the time.

Interactions with other objects

Brick wall is inanimate. It doesn't actually do anything by itself, and therefore doesn't interact with any other objects.

Specification

Basically, brick wall is just wall, with the properties that it is **rounded** (and thus boulders or diamonds can roll off it), and that unlike steel wall, it can be consumed in an explosion.

Magic wall and expanding wall look the same as brick wall most of the time.

BDCFF Object 0004: Steel wall

Object number: $0004
Game class: Boulder Dash (by Peter Liepa)
Object name: Steel wall

In this document:

Properties

Animate: no
Impact explosive: no
Chain explosion action: unaffected
Explosion type: n/a
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics

This GIF shows the graphic of steel wall from the C64 implementation of Boulder Dash (hence the graphic is 8 double-width pixels wide and 16 pixels high). The object does not have an animation sequence: it looks the same all the time.

Interactions with other objects

Steel wall is inanimate. It doesn't actually do anything by itself, and therefore doesn't interact with any other objects.

Specification

Basically, steel wall is just wall, with the property that it remains unaffected by explosions. It is not **rounded**, so boulders and diamonds sit on top without rolling off.

BDCFF Object 0005: Expanding wall

Object number: $0005
Game class: Boulder Dash (by Peter Liepa)
Object name: Expanding wall

In this document:

Properties

Animate: yes
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics

Expanding wall looks identical to brick wall.

Specification

Expanding wall looks identical to brick wall, but is not **rounded**; things don't roll off it.

Expanding wall "grows" sideways whenever possible. If ever a piece of expanding wall discovers that the position to the immediate left or right is empty space, it grows to fill that space. A piece of expanding wall can expand to fill both the left and the right spaces in one frame. However, the newly created pieces of expanding wall don't get to themselves grow until the next frame, so if you put a piece of expanding wall into a cave of space, you will see it expanding left and right at the rate of one space per frame.

Each time expanding wall grows, an appropriate sound is made (failing all else, just the standard boulder crash sound, the same as is used when a boulder starts or stops falling, or is pushed).

General Algorithm

```
procedure ScanExpandingWall(in positionType wallPosition)
# Local variables
 positionType NewPosition;

# Try to grow left
 NewPosition := GetRelativePosition(wallPosition, left1);
 if (GetObjectAtPosition(NewPosition) == objSpace) then
 PlaceObject(objExpandingWall, attribNone, NewPosition);
 endif

# Try to grow right
 NewPosition := GetRelativePosition(wallPosition, right1);
 if (GetObjectAtPosition(NewPosition) == objSpace) then
 PlaceObject(objExpandingWall, attribNone, NewPosition);
 endif
endprocedure
```

BDCFF Object 0006: Rockford

Object number: $0006
Game class: Boulder Dash (by Peter Liepa)
Object name: Rockford

In this document:

Properties

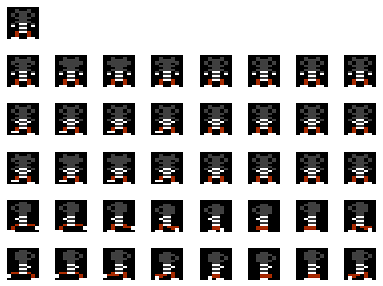
Animate: yes
Impact explosive: yes
Chain explosion action: consumed
Explosion type: explodeToSpace
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics


This GIF shows the animation sequences of Rockford from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

There are six rows of graphics:

1. Rockford facing forward (no animation sequence; just one graphic);
2. Rockford facing forward, blinking
3. Rockford facing forward, tapping foot
4. Rockford facing forward, blinking and tapping foot
5. Rockford facing left
6. Rockford facing right

Rockford Animation

This description of the Rockford animation sequences is based on the C64 implementation. Other implementations may do things differently. However, the Rockford blinking and tapping was considered an important part of the look and feel of the C64 implementation at least.

Facing left or right

There are animation sequences for when Rockford is moving left or right, but not up or down. What happens is that if Rockford is moving up or down, then the animation sequence of the last horizontal movement is used. For example, if Rockford moves right and then up, the "moving-right" animation sequence will be used for that time. If Rockford then moves left and up, the "moving-left" animation sequence is used during this time. This brings up an obvious question: what happens at the beginning of a cave if Rockford moves up or down without first having moved horizontally? The answer is that Rockford will face left as he moves up or down initially.

Tapping foot and blinking

When Rockford is not moving, he faces forward (out of the screen towards the player). Rockford gets bored; he taps his foot and blinks his eyes. Blinking and tapping are independent of each other; in the C64 implementation, the upper and lower halves of his body are controlled separately.

Animation sequences are eight frames long, and each frame is displayed for two ticks, so it takes 16 ticks (about 0.27 seconds) to complete each animation sequence. At the beginning of each new animation sequence (ie every 0.27 seconds), if Rockford is not currently moving, it is decided whether Rockford will be idle, blink, tap his foot, or blink and tap his foot. There is a 25% (1/4) chance he will blink each animation sequence, and a 6.25% (1/16) chance that he will stop tapping (if he is currently tapping) or start tapping if he isn't.

Interactions with other objects

Rockford interacts with the following objects:

- Space, Dirt: can be moved through, leaving space behind
- Diamond: can be collected, leaving space behind. Rockford is able to collect diamonds while they are falling. In particular, it is useful to hold down the fire button and collect diamonds as they fall past.
- Boulder: can be pushed. There is a 1 in 8 chance that Rockford successfully pushes a boulder each frame that he tries (and that it is possible he succeeds). Rockford can not push boulders that are falling.
- Outbox: causes the cave to be exited

Specification

Rockford is the player, and so Rockford's movements are controlled by the player. The basic objective being, of course, to collect enough diamonds to activate the exit, and to get out alive (without being killed by various means or running out of time).

Rockford movement performed during scan routine

There are three posibilities for when an implementation of BoulderDash can check the player controls (joystick) and move Rockford:

1. Before the cave scan routine
2. During the cave scan routine
3. After the cave scan routine

When the scan is performed can make a difference as to how the objects in the cave interact. If Rockford and another object both attempt to move into the same position, who succeeds depends on when the Rockford move routine is performed. If it is done before the cave scan, Rockford will always get priority. If it is done during the cave scan, who gets priority depends on the direction Rockford is moving in. And if the Rockford move routine is done after the cave scan routine, then the other object will get priority.

In BoulderDash I on the C64, the Rockford move routine is done during the cave scan routine: that is, when the scan routine comes across Rockford, it then looks at the joystick and processes it appropriately. This technique has an advantage: it doesn't presume that Rockford actually exists in the cave (Rockford doesn't exist during the pre-Rockford sequence or after he dies). Doing it this way makes it easier for the cave to continue on in its normal way regardless of whether Rockford exists or not.

Movement controls

Using the joystick or other control device, the player can control Rockford's movement. Rockford can move up, down, left and right only, not diagonally. In BoulderDash I for the C64, if the player presses diagonally, Rockford moves horizontally (the horizontal component has priority over the vertical component). Although perhaps not critical, I personally am used to the player controls behaving in this fashion, and I initially found it frustrating when playing an implementation that behaved differently. Each implementation may well have its own idiosyncrasies, but if you're out to clone the C64 look & feel, then you might want to take note of the C64's particular behaviour.

The fire button on the joystick can be used to "move without moving"; that is, everything happens as if Rockford did move in the indicated direction, but Rockford isn't actually moved. This is critical for some caves which require this technique.

General Algorithm

```
procedure ScanRockford(in positionType currentScanPosition;
 inout positionType RockfordLocation;
 inout Boolean RockfordAnimationFacingDirection;
 in Boolean demoMode;
 in Boolean twoJoystickMode;
 in playerType currentPlayer;
 inout integer numRoundsSinceRockfordSeenAlive)
# We have come across Rockford during the scan routine. Read the joystick
or
# demo data to find out where Rockford wants to go, and call a subroutine
to
# actually do it.

 ASSERT(numRoundsSinceRockfordSeenAlive >= 0);

# Local variables
 joystickDirectionRecord JoyPos;

# If we're in demo mode, we get our joystick movements from the demo data
 if (demoMode) then
 JoyPos := GetNextDemoMovement();
 else

# Otherwise if we're in a real game, we get our joystick movements from
# the current player's input device (joystick, keyboard, whatever).
 JoyPos := GetJoystickPos();
 endif

# Call a subroutine to actually deal with the joystick movement.
 MoveRockfordStage1(currentScanPosition, RockfordLocation, JoyPos,
RockfordAnimationFacingDirection);

# Rockford has been seen alive, so reset the counter indicating the number
# of rounds since Rockford was last seen alive.
 numRoundsSinceRockfordSeenAlive := 0;
endprocedure ScanRockford

##

procedure MoveRockfordStage1(in positionType currentScanPosition;
 inout positionType RockfordLocation;
 in joystickDirectionRecord JoyPos;
 inout Boolean RockfordAnimationFacingDirection)
# Note: in this routine, if the user presses diagonally, the horizontal
movement takes
# precedence over the vertical movement; ie Rockford moves horizontally.

# Local variables
 Boolean ActuallyMoved;
 positionType NewPosition;

# Determine Rockford's new location if he actually moves there (ie he isn't
# blocked by a wall or something, and isn't holding the fire button down).
 switch (JoyPos.direction) of
 case down:
 RockfordMoving := true;
 NewPosition := GetRelativePosition(currentScanPosition, down1);
 elscase up:
 RockfordMoving := true;
 NewPosition := GetRelativePosition(currentScanPosition, up1);
 elscase right:
 RockfordMoving := true;
 RockfordAnimationFacingDirection := facingRight;
 NewPosition := GetRelativePosition(currentScanPosition, right1);
 elscase left:
 RockfordMoving := true;
 RockfordAnimationFacingDirection := facingLeft;
 NewPosition := GetRelativePosition(currentScanPosition, left1);
 else
 RockfordMoving := false;
 endcase

# Call a subroutine to actually deal with this further.
 ActuallyMoved := MoveRockfordStage2(currentScanPosition, NewPosition,
JoyPos);

# If Rockford did in fact physically move, we update our record of
Rockford's
# position (used by the screen scrolling algorithm to know where to
scroll).
 if (ActuallyMoved) then
 RockfordLocation := NewPosition;
 endif
 endswitch
endprocedure MoveRockfordStage1

##

function MoveRockfordStage2(in positionType originalPosition;
 in positionType newPosition;
 in joystickDirectionRecord JoyPos):Boolean
# Part of the Move Rockford routine. Call MoveRockfordStage3 to do all the
work.
# All this routine does is check to see if the fire button was down, and
# so either move Rockford to his new position or put a space where he would
# have moved. Returns true if Rockford really did physically move.

# Local variables
 Boolean ActuallyMoved;

# Call a subroutine to move Rockford. It returns true if the movement was
# successful (without regard to the fire button).
 ActuallyMoved := MoveRockfordStage3(newPosition, JoyPos);

# If the movement was successful, we check the fire button to determine
# whether Rockford actually physically moves to the new positon or not.
 if (ActuallyMoved) then
 if (JoyPos.fireButtonDown) then
 PlaceSpace(newPosition);
 ActuallyMoved := false;
 else
 PlaceRockford(newPosition);
 PlaceSpace(originalPosition);
 endif
 endif

# Tell our caller whether or not Rockford physically moved to a new
position.
 return ActuallyMoved;
endfunction MoveRockfordStage2

##

function MoveRockfordStage3(in positionType newPosition;
 in joystickDirectionRecord JoyPos):Boolean
# See what object is in the space where Rockford is moving and deal with it
# appropriately. Returns true if the movement was successful, false
```

```
otherwise.

# Local Variables
 Boolean movementSuccessful;
 objectType theObject;
 positionType NewBoulderPosition;

# Determine what object is in the place where Rockford is moving.
 movementSuccessful := false;
 theObject := GetObjectAtPosition(newPosition);
 switch (theObject) of

# Space: move there, and play a sound (lower pitch white noise)
 case objSpace:
 movementSuccessful := true;
 RequestRockfordMovementSound(movingThroughSpace);

# Dirt: move there, and play a sound (higher pitch white noise)
 elscase objDirt:
 movementSuccessful := true;
 RequestRockfordMovementSound(movingThroughDirt);

# Diamond: pick it up
 elscase objStationaryDiamond:
 movementSuccessful := true;
 PickUpDiamond();

# OutBox: flag that we've got out of the cave
 elscase objOutBox:
 movementSuccessful := true;
 FlagThatWeAreExitingTheCave(and that we got out alive);

# Boulder: push it
 elscase objStationaryBoulder:
 if (JoyPos.direction == left) then
 NewBoulderPosition := GetRelativePosition(newPosition, left1);
 if (GetObjectAtPosition(NewBoulderPosition) == objSpace) then
 movementSuccessful := PushBoulder(NewBoulderPosition);
 endif
 elsif (JoyPos.direction == right) then
 NewBoulderPosition := GetRelativePosition(newPosition, right1);
 if (GetObjectAtPosition(NewBoulderPosition) == objSpace) then
 movementSuccessful := PushBoulder(NewBoulderPosition);
 endif
 endif
 endcase
 endswitch

# Return an indication of whether we were successful in moving.
 return movementSuccessful;
endfunction MoveRockfordStage3

##

function PushBoulder(in positionType newBoulderPosition)
# There is a 12.5% (1 in 8) than Rockford will succeed in pushing the
boulder.
# Return true if boulder successfully pushed, false if not.

# Local variables
 Boolean pushSuccessful;
 pushSuccessful := (GetRandomNumber(0, 7) == 0);
 if (pushSuccessful) then
 RequestSound(boulderSound);
 PlaceBoulder(newBoulderPosition);
 endif

 return pushSuccessful;
endfunction PushBoulder

##

procedure PickUpDiamond()
# Player has picked up a diamond. Increase their score, increase their
number
# of diamonds collected, and check whether they have enough diamonds now.

 RequestSound(pickedUpDiamondSound);
 CurrentPlayerData.score += CurrentPlayerData.currentDiamondValue;
 CheckForBonusLife();
 CurrentPlayerData.diamondsCollected++;
 CheckEnoughDiamonds();
endprocedure PickUpDiamond

##

procedure CheckEnoughDiamonds()
 if (CurrentPlayerData.diamondsCollected == CaveData.diamondsNeeded) then
 CurrentPlayerData.gotEnoughDiamonds := true;
 CurrentPlayerData.currentDiamondValue := CaveData.extraDiamondValue;
 Update statusbar;
 RequestSound(crackSound);
 Request screen to flash white to indicate got enough diamonds;
 endif
endprocedure CheckEnoughDiamonds

##

procedure AnimateRockford(inout Boolean Tapping;
 inout Boolean Blinking;
 in Boolean RockfordAnimationFacingDirection)
# Called by the animation routine every animation frame

# If Rockford is currently moving, we display the right-moving or left-
moving animation
# sequence.
 if (RockfordMoving)

# Can't tap or blink while moving
 Tapping := false;
 Blinking := false;

# Set up animation left or right as appropriate
 if (RockfordAnimationFacingDirection == facingRight)
 doing right-facing Rockford animation sequence
 else
 doing left-facing Rockford animation sequence
 endif

# If Rockford is not currently moving, we display a forward facing
animation sequence.
```

```
# Rockford might be idle, tapping, blinking, or both.
 else

# If we're at the beginning of an animation sequence, then check whether we
# will blink or tap for this sequence
 if (AnimationStage == 1)

# 1 in 4 chance of blinking
 Blinking := (GetRandomNumber(0, 3) == 0);

# 1 in 16 chance of starting or stopping foot tapping
 if (GetRandomNumber(0, 15) == 0)
 Tapping := not Tapping;
 endif
 endif

 doing forward-facing Rockford animation sequence (idle, blink, tap, or
blink&tap)

endprocedure AnimateRockford
```

BDCFF Object 0007: Dirt

Object number: $0007
Game class: Boulder Dash (by Peter Liepa)
Object name: Dirt

In this document:

Properties

Animate: no
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics

This GIF shows the graphic of dirt from the C64 implementation of Boulder Dash (hence the graphic is 8 double-width pixels wide and 16 pixels high). It does not have an animation sequence: it looks the same all the time.

Interactions with other objects

Dirt is inanimate. It doesn't actually do anything by itself, and therefore doesn't interact with any other objects.

Specification

Dirt exists just to hold things up. :) Rockford can move through dirt, amoeba can grow into dirt, but boulders and diamonds sit on dirt without rolling off. Fireflies and butterflies can not move through dirt.

BDCFF Object 0008: Firefly

Object number: $0008
Game class: Boulder Dash (by Peter Liepa)
Object name: Firefly

In this document:

Properties

Animate: yes
Impact explosive: yes
Chain explosion action: consumed

Explosion type: explodeToSpace
Rounded: no

Attributes

Attribute format: %00000000 000000aa

aa: Initial "facing" direction:

- 00 = facing left
- 01 = facing up
- 10 = facing right
- 11 = facing down

All fireflies should start life "facing left" (aa=00). This means that the first movement a fly will make is down (if possible), because the fly will turn to its left, resulting in moving down.

Graphics

This GIF shows the animation sequence of a firefly from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

Interactions with other objects

The firefly interacts with the following objects:

- Explodes on contact with: amoeba, Rockford, Rockford (scanned this frame)

Specification

Fireflies, like butterflies, are creatures which move around in a set pattern. If impacted, or on contact with amoeba or Rockford, fireflies explode into space.

Fireflies are considered to be facing in one of four directions (up, down, left, right) although visually you can't tell which direction a firefly is currently facing by looking at it. Fireflies always like to turn to their left if possible (going round and round in circles if possible), failing that they will go forward, or finally they will turn to their right if they can't move forward.

Turning corners

Fireflies do not always turn corners instantly. Flies can turn in their "preferred direction" instantly, but otherwise take time to turn against their "preferred direction".

In BoulderDash I (on the C64), fireflies always begin "facing left", meaning that they will make their first movement down if possible (because fireflies always try to turn left when possible). The C64 cave format allows you to specify the initial direction of each firefly individually, however all caves in the C64 BoulderDash I have their fireflies facing left to begin with. It is recommended that all fireflies begin life facing left.

The way a firefly works is this:

```
if (space to the firefly's left is empty) then
 turn 90 degrees to firefly's left;
 move one space in this new direction;
} else if (space ahead is empty) then
 move one space forwards;
} else {
 turn 90 degrees to the firefly's right;
 _do not move_;
}
```

The key thing to note is that if a firefly is forced to turn against its "preferred direction", it does not actually move for that frame.

The result is that when a firefly can make a left turn, it will do so "instantly", but if forced to make a right turn, it will pause for one frame before doing so. If forced to turn around (180 degrees), it will pause for two frames before going back the way it came.

Checking for amoeba and Rockford

A fly will check all four directions next to it for amoeba or Rockford *before* it attempts to move each frame. Should it find any amoeba or Rockford, it explodes on the spot. It is the fly that explodes, not Rockford or the amoeba.

Because the check is made before the fly moves, you may momentarily see the fly next to an amoeba/Rockford for one frame before the explosion happens. Note that (unusually), the firefly will also explode if next to a "Rockford, scanned this frame". By "scanned this frame" I mean that Rockford has already moved once during this scan frame, and it is marked as such so that if the same Rockford is come across again in the same scan frame (because Rockford moved down or right) then the player won't have the opportinity to move Rockford again.

General Algorithm

```
procedure ScanFirefly(in positionType positionOfFirefly;
 in directionType directionOfFirefly)
# Local variables
 positionType NewPosition;
 directionType NewDirection;

# First check whether the firefly will explode by being next to Rockford,
# Rockford-scanned-this-frame or amoeba but not amoeba-scanned-this-frame.
 if (FlyWillExplode(positionOfFirefly)) then
 Explode(positionOfFirefly, explodeToSpace);
 else

# Failing that, attempt to move turn left and move there if possible
 NewPosition = GetNextFlyPosition(positionOfFirefly, directionOfFirefly,
turnLeft);
 if (GetObjectAtPosition(NewPosition) == objSpace) then
 NewDirection = GetNewDirection(directionOfFirefly, turnLeft);
 PlaceFirefly(NewPosition, NewDirection);
 PlaceSpace(positionOfFirefly); # ie old position
 else

# Failing that, attempt to move straight ahead
 NewPosition = GetNextFlyPosition(positionOfFirefly, directionOfFirefly,
straightAhead);
 if (GetObjectAtPosition(NewPosition) == objSpace) then
 PlaceFirefly(NewPosition, directionOfFirefly); # ie keep same direction
 PlaceSpace(positionOfFirefly); # ie old position
 else

# Failing that, turn to the right but do not move
 NewDirection = GetNewDirection(directionOfFirefly, turnRight);
 PlaceFirefly(positionOfFirefly, NewDirection); # old position, new
direction
 endif
 endif
 endif
endprocedure

##

function FlyWillExplode(in positionType aPosition):Boolean
# Check the four directions around a fly at a given position to see whether
# it will explode. Returns true if so, false if not.

# Local variables
 Boolean ExplodedYet;

# Check the four directions to see whether the fly will explode
 ExplodedYet := CheckFlyExplode(GetRelativePosition(aPosition, up1));
 if (not ExplodedYet) then
 ExplodedYet := CheckFlyExplode(GetRelativePosition(aPosition, left1));
 endif
 if (not ExplodedYet) then
 ExplodedYet := CheckFlyExplode(GetRelativePosition(aPosition, right1));
 endif
 if (not ExplodedYet) then
 ExplodedYet := CheckFlyExplode(GetRelativePosition(aPosition, down1));
 endif
```

```
# Return function result
 return ExplodedYet;
endfunction

##

function CheckFlyExplode(in positionType aPosition):Boolean
# Check the given position to see whether it contains an object which a
# fly will explode if it is in contact with (ie Rockford or Amoeba).
# Returns true if so, false if not.

 return (GetObjectAtPosition(aPosition) in {objRockford,
objRockfordScanned, objAmoeba});
endfunction
```

BDCFF Object 0009: Butterfly

Object number: $0009
Game class: Boulder Dash (by Peter Liepa)
Object name: Butterfly

In this document:

Properties

Animate: yes
Impact explosive: yes
Chain explosion action: consumed
Explosion type: explodeToDiamonds
Rounded: no

Attributes

Attribute format: %00000000 000000aa

aa: Initial "facing" direction:

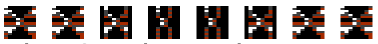
- 00 = facing left
- 01 = facing up
- 10 = facing right

- 11 = facing down

Note: These codes are *different* than those used internally by the Commodore 64 implementation of Boulder Dash. The above codes are the same as for the firefly, for consistency, rather than the C64 implementation which uses 00=down, 01=left, 10=up, 11=right.

All fireflies should start life "facing down" (aa=11). This means that the first movement a butterfly will make is left (if possible), because the fly will turn to its right, resulting in moving left.

Graphics


This GIF shows the animation sequence of a butterfly from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

Interactions with other objects

The butterfly interacts with the following objects:

- Explodes on contact with: amoeba, Rockford, Rockford (scanned this frame)

Specification

Butterflies are the same as fireflies except

- butterflies usually begin life facing down rather than left
- butterflies prefer to go to their right (failing that go straight, failing that turn left)
- they look different
- when they explode, they explode to diamonds rather than space

Please refer to the firefly specification for full details.

BDCFF Object 000A: Amoeba

Object number: $000A
Game class: Boulder Dash (by Peter Liepa)
Object name: Amoeba

In this document:

- Properties
- Attributes
- Graphics
- Interactions with other objects
- Specification
- General Algorithm

Properties

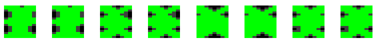
Animate: yes
Impact explosive: no
Chain explosion action: consumed
Explosion type: n/a
Rounded: no

Attributes

Attribute format: %00000000 00000000

There are no attributes for this object type.

Graphics


This GIF shows the animation sequence of amoeba from the C64 implementation of Boulder Dash (hence the graphics are 8 double-width pixels wide and 16 pixels high).

Interactions with other objects

The boulder interacts with the following objects:

- Objects it can grow into: space, dirt

Specification

Amoeba is stuff that grows randomly. If trapped such that it can't grow any more, it "suffocates" and turns into diamonds. If it grows too large, it turns into boulders. Fireflies and butterflies will explode on contact with amoeba.

Every scan, a count is kept of how many amoeba have been found. For each amoeba found during the current scan, it does these things:

1. If there were too many (see below) amoeba found in the scan during the *last* frame, the amoeba is considered to have grown too large, and so all amoeba found in *this* scan frame are quietly replaced with boulders.
2. Failing that, if it was determined in the scan during the *last* frame that the amoeba was completely enclosed (could not grow), then each amoeba is quietly replaced with a diamond.
3. Failing that, if there have been no amoeba found during the *current* scan that had the potential to grow, then a check is made to see whether this amoeba could grow. If it is possible for it to grow, then the flag is changed to indicate that there is at least one amoeba in existance that can grow during this frame.
4. If the amoeba did not turn into a diamond or a boulder (in steps 1 or 2 above), it may or may not attempt to grow. A random number is generated to decide whether the amoeba will attempt grow: it has a 4/128 chance (about 3%) normally, or a 4/16 chance (25%) in some circumstances. If the decision is that the amoeba will atempt to grow, it randomly chooses one of the four directions to grow in. If that direction contains a space or dirt, the amoeba grows to fill that spot. The new amoeba just grown does not itself get the chance to grow until the next frame (ie the new amoeba is marked as "amoeba, scanned this frame").

How many is too many?

For the Commodore 64 implementation of Boulder Dash, "too many" amoeba (the point where they turn into boulders) is 200 or more. Since other implementations of Boulder Dash may permit cave sizes other than 40 x 22 (= 880 squares), I suggest that "too many" is defined as being 200/880 = 22.7% of the total number of squares available in the cave. In other words, once 22.7% or more of the cave is occupied by amoeba, it should turn into boulders.

When is it 3% and when 25%?

Initially, the amoeba growth probability is 4/128 (about 3%). Once the "amoeba slow growth time" has elapsed, the amoeba suddenly starts growing a lot quicker (amoeba growth probability = 25%). The "amoeba slow growth time" is set on a cave-by-cave basis, and is in seconds.

General Algorithm

```
procedure ScanAmoeba(in positionType positionOfAmoeba;
 in integer anAmoebaRandomFactor;
 in Boolean amoebaSuffocatedLastFrame;
 inout Boolean atLeastOneAmoebaFoundThisFrameWhichCanGrow;
 in integer totalAmoebaFoundLastFrame;
 inout integer numberOfAmoebaFoundThisFrame)
# Local variables
 directionType direction;
 positionType NewPosition;

 ASSERT(anAmoebaRandomFactor > 0);
 ASSERT(totalAmoebaFoundLastFrame > 0);
 ASSERT(numberOfAmoebaFoundThisFrame > 0);
 numberOfAmoebaFoundThisFrame++;

# If the amoeba grew too big last frame, morph into a boulder.
# kTooManyAmoeba = 200 for original Boulder Dash.
 if (totalAmoebaFoundLastFrame >= kTooManyAmoeba) then
 PlaceObject(objBoulder, attribStationary, positionOfAmoeba);
 else

# If the amoeba suffocated last frame, morph into a diamond
 if (amoebaSuffocatedLastFrame) then
 PlaceObject(objDiamond, attribStationary, positionOfAmoeba);
 else

# If we haven't yet found any amoeba this frame which can grow, we check to
# see whether this particular amoeba can grow.
 if (not atLeastOneAmoebaFoundThisFrameWhichCanGrow) then
 foreach direction in (up1, left1, right1, down1) do
 if (GetObjectAtPosition(GetRelativePosition(positionOfAmoeba, direction))
in {objSpace, objDirt}) then
 atLeastOneAmoebaFoundThisFrameWhichCanGrow := true;
 endif
 endforeach
 endif

# If this amoeba decides to attempt to grow, it randomly chooses a
direction,
# and if it can grow in that direction, does so.
 if (AmoebaRandomlyDecidesToGrow(anAmoebaRandomFactor)) then
 direction := GetRandomDirection();
 NewPosition = GetRelativePosition(positionOfAmoeba, direction);
 if (GetObjectAtPosition(NewPosition) in {objSpace, objDirt}) then
 PlaceObject(objAmoeba, attribNone, NewPosition);
 endif
 endif
 endif
 endif
endprocedure

##

function AmoebaRandomlyDecidesToGrow(in integer
anAmoebaRandomFactor):Boolean
# Randomly decide whether this amoeba is going to attempt to grow or not.
# anAmoebaRandomFactor should normally be 127 (slow growth) but sometimes
is
```

```
# changed to 15 (fast growth) if the amoeba has been alive too long.
 ASSERT(anAmoebaRandomFactor in {15, 127});
 return (GetRandomNumber(0, anAmoebaRandomFactor) < 4);
endfunction
```
