State Machines Programming*

Malleus <negrente@arena.sci.univr.it>

June 28, 2000

1 The programming language

1.1 General Structure

The programming language used to bulid state machines is a block structure language. Like C or C++, spacing characters are space, tab and return. You can write comments to the code including them into `/*` and `*/`. A set of predefined commands represent the keywords of the language. Each state machine is fully described by a set of commands built into a `STATE_MACHINE {}` block. That block contains two sections:

- one section containing informations only for QUEDITOR;
- one code section, that describes the state machine's logic. That section, delimited by the `CODE []` command, is used directly by the game itself. QUEDITOR does not analize in any way that code.

1.2 QUEDITOR informations

1.2.1 State machine's name

QUEDITOR uses that name in the state machines'selection list (*Create new machine* window). The declaration is made using the `KEY {<name>}` instruction.

1.2.2 Short description

QUEDITOR uses that information in the *Create new machine* and *Machine properties...* dialogue windows. The declaration of the description is made using the `COMMENT [<description>]` instruction.

*Translation of *Khelb*'s *Programmation des machines à états*, Chapter 4, *Le langage de programmation*. You can find the original document at *Le Ouaibe @ Khelburn*: `http://perso.wanadoo.fr/dubois.pascal/`, or send email to *Khelb* himself: `<khelburn@wanadoo.fr>`.

1.2.3 Parameters

Those parameters can be modified by the *Machine properties...* dialogue window. Each parameter is described by a `PARAM {}` block, containing the following commands:

- an id-number key (`KEY {}` command) that can be referenced from within the code section (syntax `%<num>`). The id-number key 0 is predefined by QUEDITOR and designs the *unique name* of the machine's instance (*key* field of *Create new machine* window).
- A parameter's description (command `COMMENT [<description>]`).
- A type (command `TYPE <type>`), that can be:
  - `ALPHA` means a characters'string.
  - `FRAME` means a graphic object (one of the possible graphic representations of the machine).
  - `INT` means an integer value.
  - `LIST` means a list of values. If the `LIST` command is used, each value the parameter could have is declared by a `LIST [<description> = <value>]` command. When a `LIST` command is used, the `%<num>` reference assumes the value of `<value>`.
  - `MONSTER` means a monster.
  - `OBJECT` means an object.
  - `SKELFILE` means the *squelette* file of a 3D object. A *squelette* file contains the graphic representation (`SKELNAME`) and the animations (`SKELANIM`) of a graphic object.
  - `SKELNAME` means the graphic representation extracted by a *squelette* file. QUEDITOR gives to the `SKELFILE` parameter the *same value* as `SKELNAME`.
  - `SOUND` means a sound.
  - `TEXT` means a text.
- A default value (`DEFAULT {<value>}` command). This parameter is optional, but it's recomanded for a `LIST` type, where `<value>` has to be a valid `<description>`.
- Some complementary informations, depending on the parameter:
  - `DISPLAY {}`: used with a `FRAME` or `SKELNAME` type, defines[1] the object used by QUEDITOR to represent this machine, both in hierarchy and 3D view. Each state machine must have a `DISPLAY {}`

[1] Hereafter, I used the sans serifs font to insert personal comments or when I was uncertain on the translation.

command in order to use it. I suggest you to not use more than one DISPLAY {} command for each machine: once I had two DISPLAY {} commands in the same machine, and it acted strange, esp. when saving and loading quest with QUEDITOR.

- IN {}: used with an OBJECT or MONSTER type, allows to view the objects and/or the monsters *used* by the machine in the hierarchy view.
- LIST [<description> = <value>]: used with a LIST type describes one of the admissible values to be used by the parameter.
- OUT {}: used with an OBJECT or MONSTER type, allows to view the objects and/or the monsters *created* by the machine in the hierarchy view.

1.3 Code section

Note: each state machine has an *action zone* of 30x30 squares. The machine is activated (it executes its code) when a character enters this zone, and is deactivated (it stops executing its code) when all characters have left that zone.

1.3.1 General structure

A state machine's code section contains a unique TRAP {} block, which contains:

- a *standard* KEY {%0} command, that allows DARKSTONE to identify precisely that state machine. Remember that %0 id-number key is predefined by QUEDITOR and defines the unique name for that particular machine's instance (*key* field of *Create new machine* window).
- a FLAGS {<flags>} command, that defines the machine's properties. <flags> is a list of keywords spaced by the " | " chracter, in any order. The keywords are:
  - ACTIVE: makes the machine usable in an interactive way. The characters can't perform any interactive action on a machine with no ACTIVE flag. A machine without this flag makes the mouse pointer not to turn red when it passes over it.

    Note: a machine becomes automatically *inactive* once it has passed every possible state; in order to make a *permanent* machine (ie. a machine that keeps working, and is not automatically deactivated) the machine must have a state that can't be reached (ie. a state which is never addressed by any GOTO {} or JUMP {} instruction).

- `ATTACK`: the characters approach the machine and *attack* it when you click on it.
- `CHANGELEVEL`: means that the machine implies a change of dungeon's level.
- `CLEARMOUSE`: the machine deletes the object on the mouse when it performs a transition based on the object's detection (cfr. Section 1.3.2).
- `DROP`: allows the drop of an object on the machine's graphic representation.

  Note: it's still possible to drop an object on a machine with no graphic representaion, even if it has no `DROP` flag.
- `HANDLE`: ???
- `LIGHT`, `LIGHTBUFFER`: when both used, they cause the machine to light up at the mouse's passage.
- `NEEDPOS`: if used, when you click on the machine, the characters approach it from the *front*[2] side; otherwise they'd approach it from any side.
- `NOSHADOW`: suppress the shadow of animated frames (see `SKELNAME` and `SKELANIM`, Section 1.3.4).
- `POSEND`: prevents the selection of the machine location as a displacement target (especially useful for doors, not to get "stucked" in the graphics).
- `SIDEDETECT`: ???
- `STAIR_DOWN`: the machine causes a forward transition to a lower dungeon (higher difficulty). Each dungeon level should have one and only one machine with this flag.
- `STAIR_UP`: the machine causes a backward transition to a higher dungeon (lower difficulty). Each dungeon level should have one and only one machine with this flag. You have to make these machines only if you build an entire level; you can't build a room in a dungeon with a *stair* machine – ie. you have to let the QUEDITOR's random engine to do that work.
- `TRANSMIT`: the machine can send and receive signals.

- Some `STATE` blocks, describing how the machine works in each state. *The first state indicates the starting state of the machine.* Common commands to the whole set of states, or indipendent from any particular state, can be placed before the declaration of the first state.

[2]The *front* of a 3D frame should be checked for each frame: for example, the *front* side of the demon's statue is in fact its *back* side...

1.3.2 State's description (`STATE {}` block)

Each state contains:

- a name, unique for each state in the same machine, declared with the `KEY {<statename>}` command. That name is used in the states'transition (`GOTO {<statename>}` and `JUMP {<statename>}` commands).
- An instruction set executed each time the machine enters into that state. These instructions are described in Section 1.3.4.

  Note: the instructions are executed before entering into the `MULTI {}` block (see below).
- A state transition, event-based, logic:
  - events that cause a state transition are described into a `MULTI {}` block.
  - each event is described into a `CONDITION {}` block, containing two commands. The former describes the event that causes the transition, the latter is a `GOTO {<name of the state>}` command that indicates the machine's next state. The events that could trigger a transition are described in Section 1.3.3.
  - the first event (in the list order) that occours (evalued as *true*) triggers the transition to the state described by the corresponding `GOTO {<name of the state>}` command. If no transition is possible, the machine waits in the current state.

1.3.3 Events

The events that can be used into a `CONDITION {}` block to trigger a transition are:

- `CLICK {}`: *true* if the player left-clicks on (the graphic representation[3] of) the machine.

  Note: if the machine has the `ATTACK` flag set, the characters would "attack" the machine – ie. they would swing their weapon onto the machine.
- `CLOSEPANEL {}`: *true* if the player closes a text panel, previously opened with the `TEXTPANEL {<text>}` command.
- `COLLID {<collision>}`: *true* if a character, a monster or an object take up (`<collision> = 1`) the machine's place; or anything does not take up (`<collision> = 0`) the machine's place.

[3]Hereafter, clicking on, touching, attacking, etc. the state machine has to be intended as clicking on, touching, attacking, etc. *the machine's graphic representation.*

- `COLLIDEX {<collision>, <distance>, <detection>}`: *true* if the parameter indicated `<detection>` is placed (`<collision> = 1`) or is not placed (`<collision> = 0`) at less than `<distance>` from the machine's place. The `<detection>` parameter can be one of these values:

  1: a character;

  2: an object;

  3: a character or an object;

  4: a monster;

  5: a character or a monster;

  6: an object or a monster;

  7: a character, an object or a monster.

  Note: `COLLID {<collision>}` and `COLLIDEX {<collision>, 1, 7}` are just the same.

- `ENDANIM {}`: *true* when the animation caused by a `SKELANIM` instruction ends.

- `MONSTERCOUNT {<monster>, <num>}`: *true* if the number of monsters of type `<monster>` into the current dungeon/land level is strictly lower than `<num>`.

- `MSG {<#msg>}`: *true* if the machine receives the signal number `<#msg>`.

- `NOPANEL {}`: always *true*. It turns *false* if a text panel (`TEXTPANEL {<text>}` command) is opened.

- `OBJECT {<object>}`: *true* if the object indicated by `<object>` is on the mouse and the player left-clicks on the machine.

  Note (1): if you click on the machine with no object, or with a wrong object, the `FAIL {<text>}` or the `FAILPANEL {<text>}` instruction, which follow the `MULTI {}` block, are executed.

  Note (2): In order to make the machine perform another action, instead of simply displaying a text, you have to insert a `CLICK {}` command after the `OBJECT {<object>}` command in the `CONDITION {}` block.

- `OBJECTHAND {<object>}`: *true* if the character wears the object `<object>` on the hand(s) when the player left-clicks on the machine.

- `OBJECTINVENT {<object>}`: *true* if the character holds the object `<object>` in the inventory when the player left-clicks on the machine.

- `OBJECTWEAR {<object>}`: *true* if the character wears the object `<object>` when the player left-clicks on the machine.

- `PLAYEROBJECT {<object>, <location>}`: *true* if the character holds the object `<object>` on the location defined by `<location>` when the player left-clicks on the machine. The `<location>` parameter can be one of these values:

  0: on the mouse;

  1: on the hand(s);

  2: in the equipment;

  3: in the inverntory.

- `PROJECTILEHIT {}`: *true* if a projectile hits the machine.

- `SENSEOBJ {<object>}`: *true* if the object `<object>` is placed on the machine.

- `SENSEOBJMAGIC {<object>}`: *true* if the object placed on the machine belongs to the right object's class. The defined object's classes are defined in Table A.1.

- `SENSEWEAR {<object>, <distance>}`: *true* if the character holds the object `<object>` in the equipment while it's at less than `<distance>` from the machine.

- `TIMER {<delay>}`: *true* when `<delay>` milliseconds are passed.

  Note: `TIMER {0}` is always *true* and causes a state transition instantly.

- `VEQ {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is equal to `<value>`.

- `VGE {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is greater or equal to `<value>`.

- `VGT {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is strictly greater than `<value>`.

- `VLE {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is less or equal to `<value>`.

- `VLT {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is less than `<value>`.

- `VNE {<vobject>, <value>}`: *true* if the virtual object `<vobject>` is not equal to `<value>`.

1.3.4 Instructions

The instructions that can be used in the `CODE []` block are:

- `ACTION {}`: declares the execution of the instructions used inside that block. The available *actions* are:
  - `CAMERAFROM`, `CAMERATO` and `CAMERATIME`: change the camera position. These instructions can't be used in the quests made with QUEDITOR.
  - `CHARGECRISTAL {}`: charges the *Time Orb* with a single charge.
  - `EFFECT {<effect>}`: displays the visual effect described by `<effect>`. See Table A.2 for an effects'list and explanation.
  - `FMV {<video>}`: plays the video sequence indicated by `<video>`. The video sequence has to be placed into the *darkstone\mdata* folder. The video format should be:
    - Audio: PCM, 22,050 Hz, 16 bit, Stereo
    - Video: 352 x 232, 24 bit, 25.000 frames/sec, Indeo® video 5
  - `HIT {<damage>}`: inflicts up to `<damage>` damage points to any creature on the machine.
  - `HITEX {<damage>, <distance>}`: inflicts up to `<damage>` damage points to any creature which is at less than `<distance>` from the machine.
  - `MONSTER {<monster}`: generates the `<monster>` monster.
  - `MSG {<#msg>}`: sends the message number `<#msg>` to all linked machines.
  - `OBJECT {<object>}`: creates and places the `<object>` object on the ground.
  - `OBJECTMOUSE {<object>}`: creates and places the `<object>` object on the mouse.
  - `PLAYSONG {<numsong>}`: plays the sound sequence number `<numsong>`. *Example:* `PLAYSONG {<26>}` plays *Audren*'s song.
  - `PROJECTILE {}`: that block defines the behaviour of a projectile launched by the machine. The instructions are:
    - `KEY {<type>}`: type of the projectile launched. See Table A.3 for a projectile types'list and explanation.
    - `DAMAGEMIN {<damage>}`: minimum number of damage points that the projectile inflicts.
    - `DAMAGEMAX {<damage>}`: maximum number of damage points that the projectile inflicts.
    - `ROTATION {<val>}`: projectile's direction:

0: towards north[4];

1: towards east;

2: towards south;

3: towards west.

Note: QUEDITOR default orientation looks towards south. I've found out it's easier to build launchers in a different way:

- use a frame, like the "crossbow" one, for the machine;
- set the ROTATION value to 2: this way the crossbow fires from the front side;
- rotate the crossbow (ie. rotate the frame in 3D view) instead of changing the ROTATION value.

* YOFFSET {<val>}: height of the projectile from the ground (-140 $\leq$ <val> $\leq$ 140; where -140: ground height; 0: middle height level; 140: higher height).

– REMOVEOBJECT {<vobject>}: deletes the virtual object <vobject>.

– REMOVEOBJECTMOUSE {<object>}: deletes the object <vobject> from the mouse.

– SETLOADING {<sequence>}: defines the next loading sequence to be used. <sequence> can be one of the following:

0: stairs;

2: woods;

4: well;

6: ladder.

Direction (up/down) given by a STAIR_DOWN or STAIR_UP flag.

– sound {<sound>}: plays the sound described by <sound>.

– TELEPORT2ND {}: teleports the AI controlled character near the active character (solo mode).

– VADD {<vobject>, <val>}: adds the value <val> to the virtual object <vobject>.

– VSET {<vobject>, <val>}: sets the value <val> to the virtual object <vobject>.

- CHANGELEVEL {0}: changes level, playing the selected video sequence in function of SETLOADING, STAIR_DOWN e STAIR_UP. The new level is automatically determined.

- CLEARCOLLID {<collision>}: removes all collisions[5] on the machine. If <collision> = 0, then no collision is removed.

[4]Inverted in the original version by *Khelb*.

[5]Note by *Khelb*: You can specify a value greater than 1 in CLEARCOLLID, SETCOLLID and SETLOWCOLLID. This value is the size of the collision area. You have to be careful with that, since I noticed that in many cases, machines placed under other machines'collisions are not executing their code anymore!

- FRAME {<frame>}: displays the graphic 3D frame for the machine, using the <frame> parameter.
- JUMP {<statename>}: causes a transition towards the <statename> state of the machine.
- MSG {<#msg>}: sends the message number <#msg> to all linked machines.
- SETCOLLID {<collision>}: sets collisions on the machine. If <collision> = 0, then no collision is placed.
- SETLOWCOLLID {<collision>}: sets low collisions on the machine. If <collision> = 0, then no collision is placed.
- SKELANIM {<animation>}: causes the animation of the machine (esp. npc). <animation> is the animation model, depending on the graphic representation considered (for npc, valid values are static and attack1).
- SKELFILE {<file>}: *squelette* file containing the graphic representation for the machine.
- SKELNAME {<model>}: description model of the graphic representation for the machine (linked to SKELFILE).
- TEXT {<text zone>}: displays a text on the bottom of the screen (above the status bar). <text zone> is declared by a set of couple: <language code> {<text>}, where <language code> can be:

  0: French;

  1: English;

  2: German;

  3: Spanish;

  4: Sweden;

  5: Italian;

  6: Other.

  and <text> the text displayed.

  Note: QUEDITOR automatically formats text zones when you use a TEXT variable. The syntax is then TEXT {%<num>}.
- TEXTPANEL {<text zone>}: displays a text on a text panel (*cfr.* TEXT).

A Tables

A.1 Defined objects'classes

Object classes used in the `SENSEOBJMAGIC {<object>}` command (see 1.3.3).

Table A.1: Defined objects'classes

AMULET	Amulet
ARMOR or ARMOUR	Armour
AXE	Axe
BOW	Bow
CLUB	Club
CRISTAL	Crystal
DAGGER	Dagger or throwing dagger
ELIXIR_DEXTERITY	Dexterity potion
ELIXIR_MAGIC	Magic potion
ELIXIR_STRENGTH	Strength potion
ELIXIR_VITALITY	Vitality potion
FOOD	Food
HAMMER	Hammer
HELM	Helm
MACE	Mace
PARCHEMIN	Scroll
POTION_ANTIPOISON	Antidote potion
POTION_HEALING	Healing potion
POTION_MANA	Mana potion
POTION_POISON	Poison potion
QUEST	Quest object (weapon, armour, etc., except potions, scrolls, rings, amulets)
RING	Ring
SHIELD	Shield
STAFF	Staff
SWORD	Sword
TORCH	Torch
WEAPON	Weapon

A.2 Effects

Effects used in the `EFFECT {<effect>}` command (see 1.3.3). The descriptions of the effects are the same I used in my machines[6].

Table A.2: Effects

absorbma	Absorption Attack
absorbvi	Absorption Spell
antipoison1	Antidote No Sound
antipoison	Antidote
armormag	Light
arrowflame	Flaming Arrow
barrierea	Death Dome A
barriereb	Death Dome B
barrierec	Death Dome C
benedict	Bless
berserke	Berserker
bloodtrace	Bloodtrace
brille	Firefly
bteleport	Teleport B
confusion	Confusion
crystal	Crystal
detection	Detection
dexterity1	DEX Potion
eclaira	Spark Globe
eclairb	Spark
eclairc	Spark Explode
endarrow	End Arrow
endshurin	End Shurinken
etincel	Metal Spark
explode	Explode
farine	Flour
fear	Fear
fee2	Fee2 - Green - No Sound
fee	Fee - White
fireball2a	Fireball Red Smoke 2
fireball2b	Fireball Flash 2
fireball2c	Fireball Explode 2
fireball3a	Fireball Red Smoke 3
fireball3b	Fireball Red Smoke 3
fireball3c	Fireball Explode 3
fireballa	Fireball Red Smoke
fireballb	Fireball Flash
fireballc	Fireball Explode
firecamp	Firecamp
firedraak	Draak's Breath
firewalla	Wall of Fire A

[6]My machines are available here: `http://arena.sci.univr.it/~negrente/dsm/` .

Table A.2: Effects (continued)

firewallb	Wall of Fire B
firewall	Wall of Fire C
flamewavea	Flamethrower - Sound
flamewaveb	Flamethrower - Little Flame
flamewavec	Flamethrower - Big Flame
flash1	Thunder 1 Purple
flash2	Thunder 1 Purple Big
flash3	Thunder 2 Blue
flash4	Thunder 1 Purple 3 Blue
flash5	Thunder 1 Golden 4 Blue
flash	Flash
food	Food
fumee	Smoke
haste	Haste
heal1	Healing
ice	Ice Smoke
inberserker	Berserker Smoke
inferno	Inferno
inflame	Flame Smoke
infravision	Night Vision
invisibl	Invisibility
lenteur	Slowness
loading	Loading
magic1	MAG Potion
magicbomba	Magic Bomb Globe
magicbombb	Magic Bomb Smoke
magicbombc	Magic Bomb Explode
magicmissa	Magic Missile
magicmissb	Magic Missile Flash
magicmissc	Magic Missile Explode
maglight	Magic Light
mana1	Mana
nostone	Stone explode
nova	Time Orb
oubli	Forgetfulness
plumes	Plumes
poison1	Poison Potion
poisoned	Poisoned
poison	Poison Cloud
poisonweapon	Poison Weapon
rain	Rain
reflection2	Reflection Shield
reflection	Reflection Spell
rejuv1	Youth Potion
resurect	Resurrection
rip	Fear Single
stoncure	Stone Break

Table A.2: Effects (continued)

strength1	STR Potion
talentl	Skill Learnt
telekinesis	Telekinesis
teleport	Teleport
tempest	Storm
tfirewall	Wall of Fire Yellow Smoke
tmagicbomb	Magic Bomb Yellow Smoke
townportala	Magic Door Open
townportalc	Magic Door Enter
transform1	Mutation Spell
transform2	Mutation Target
vitality1	VIT Potion
water1	Water 1
water2	Water 2
werewolf	Werewolf

A.3 Projectiles

Projectiles used in the `KEY {<type>}` field of the `EFFECT {<effect>}` command (see 1.3.3).

Table A.3: Projectiles

`barriere`	Death Dome
`dague`	Throwing Knife
`eclair`	Spark
`eclair2`	Spark2
`fireball`	Fireball
`firewall`	Torch
`flamewave`	Flamethrower
`fleche`	Arrow
`hachette`	Throwing Axe
`magicbomb`	Magic Bomb
`magicmiss`	Magic Missile
`shuriken`	Shuriken