

С Новым годом!

Домашнее задание (для разобравшихся желающих): написать арканойд - ранее уже было рассмотрено, как сделать большую его часть, осталось разобраться, как сделать управляемую платформу и как учитывать столкновения шарика с платформой и блоками, что разобрано ниже. Скорее всего я в общих чертах расскажу, как его можно доделать, на первом занятии в новой четверти.

Управление платформой

Платформу можно рисовать по-разному, но в простейшем случае будем считать её прямоугольником. Так как она движется только по горизонтали, то достаточно ввести одну переменную - координату платформы по оси X - **var xp: integer;** при желании можно ввести и переменную для положения по оси Y , однако её значение будет постоянно, и его можно указывать просто числом. Будем считать, что координата платформы - это координата её середины. Тогда платформа рисуется так:

```
with image1.canvas do
begin
  brush.color := clGreen;
  pen.color := clYellow;
  rectangle(xp - 50, image1.Height - 20, xp + 50, image1.height - 5);
end;
```

Для движения платформы, так же, как и для движения шарика, будем рисовать её цветом фона (стирать), затем менять координату и рисовать заново в смещённом положении. Можно привязать движение платформы к разным действиям пользователя: к вращению колёсика мыши, движениям мышки, нажатию каких-либо кнопок и т.п. Я предлагаю сделать движение платформы по нажатию клавиш, чтобы разобраться с использованием клавиатуры, однако если вам кажется удобнее какой-то другой способ - можете сделать его.

Итак, простейший способ сделать движение по нажатию клавиш - это сдвигать платформу в соответствующей процедуре. Это событие есть у многих объектов, и называется оно **OnKeyDown** - по нажатию клавиши. Если объект, для которого прописано это событие, находится в фокусе, то при нажатии любой клавиши будет выполняться эта процедура, причём в ней можно будет использовать целочисленную переменную **Key**, значение которой будет соответствовать коду нажатой клавиши.

Объект находится в фокусе, если он использовался до этого. Так, например, если последним действием пользователя было нажатие кнопки (Button) или передвигание ползунка TrackBar (например ползунок регулировки скорости), то активна будет соответствующая кнопка или соответствующий TrackBar. При этом "пассивные" компоненты, например Image, не могут находиться в фокусе, и соответственно у них не предусмотрено такое событие. Фокус можно передать на какой-либо объект командой **SetFocus**, например **button1.SetFocus**; передаст фокус кнопке.

Кроме того можно фиксировать нажатия клавиш для программы в целом, независимо от того, какой компонент находится в фокусе. Для этого нужно изменить свойство формы **KeyPreview** на **True**, тогда при нажатии клавиш будут *сначала* срабатывать процедуры KeyDown, KeyPress или KeyUp формы, а *потом* - процедуры компонента, находящегося в фокусе. Порядок важен, так как он позволяет до вызова процедуры компонента изменить значение **Key** в процедуре формы, в результате чего компоненту будет казаться, что нажата другая клавиша. А если установить **Key:=0**; то компоненту будет казаться, что кнопка не нажата.

Таким образом есть два варианта сделать так, чтобы процедура движения выполнялась устойчиво: либо её нужно прописать для всех активных компонентов, либо нужно использовать переадресацию событий на форму. В обоих случаях фактически процедуру нужно написать для формы. Эта процедура, как обычно, создаётся двойным щелчком по полю напротив её названия (OnKeyDown) (см рис. 1) в инспекторе объектов. В процедуре, как уже было сказано, можно использовать переменную Key. Если была нажата стрелка влево, то она равна 37, а если стрелка вправо - 39 (*подключив библиотеку LCLType можно использовать константы vk_left и vk_right соответственно*). Таким образом процедура может быть написана так:

if (key = 37) then

begin

сдвиг влево: стирание, уменьшение координаты, рисование заново

end;

if (key = 39) then

begin

сдвиг вправо: стирание, увеличение координаты, рисование заново

end;

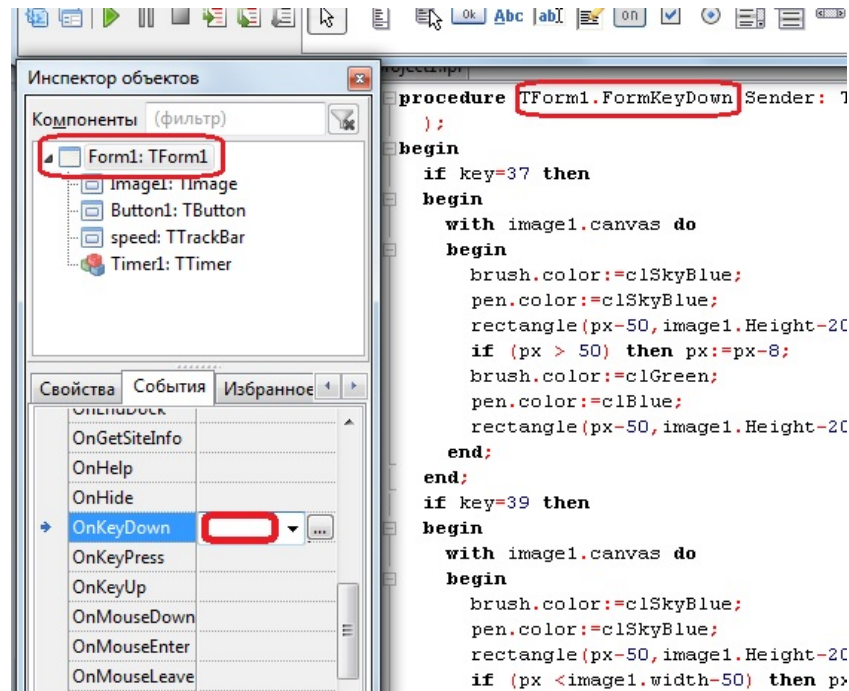


Рис. 1: Создание процедуры FormKeyDown двойным щелчком во вкладке событий.

При этом при уменьшении координаты нужно проверять, что она не слишком маленькая (> 50), а при увеличении - не слишком большая ($< \text{image1.width} - 50$), чтобы платформа не уехала за край экрана. Шаг изменения координаты **xp** определяет скорость платформы.

Осталось сделать так, чтобы эта процедура выполнялась всегда, когда нужно, для чего, как сказано выше есть два способа. Первый - просто распространить действие этой процедуры на все компоненты. Для этого необязательно писать её заново для каждого из них - можно просто выбрать уже написанную для формы процедуру в выпадающем списке (см. рис. 2).

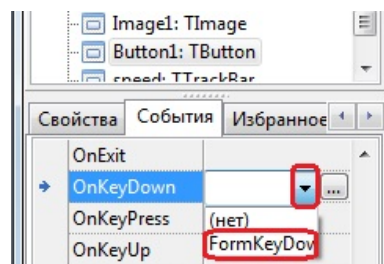


Рис. 2: Дублирование процедуры.

Однако если разобраться, то лучше использовать второй способ. Для этого нужно изменить свойство формы **KeyPreview** на **True**, а в конец процедуры приписать **Key:=0**. Это будет действовать не настолько очевидным образом, как первый способ, но зато более компактно, и позволяет при необходимости прописать другие действия для компонентов по нажатию других клавиш.

P.S. Подробнее про процедуры работы с клавиатурой, отличие их между собой и т.п. можно почитать например [здесь](#).

P.P.S. У данного способа есть один минус – при нажатии клавиши сначала срабатывает однократное нажатие (сдвиг на один шаг), а непрерывное движение начинается после небольшой паузы. Это связано с системными настройками ввода (такую же картину можно увидеть, если в текстовом редакторе держать клавишу). Чтобы избежать этого, нужно при нажатии клавиши (KeyDown) запоминать, что движение должно происходить (например в переменную типа boolean), при отпускании (KeyUp) - возвращать переменную в начальное состояние, а само движение реализовывать в процедуре таймера (с условием, что запомнено, что клавиша нажата).

Столкновение с платформой

В арканойде шарик не должен отражаться от нижней стенки. Для этого достаточно убрать из рассмотренных ранее вариантов отражение от неё, а так как условие отражения от нижней стенки по сути является условием проигрыша, то в тот же **if (...)**, который раньше и обеспечивал это отражение, можно написать например вывод в метку фразы о проигрыше и перезапуск игры.

Для самого же столкновения с платформой нужно проверять сразу несколько условий, что неудобно делать вложенными **if**-ми. Проще использовать объединение условий. Для этого используются так называемые логические операторы, из которых наиболее часто используются **and**, **or** и **not**. Они позволяют написать вместо одного условия, например $(a > 5)$ или $(b \leq 15)$ – различные их комбинации. Так оператор **not** – это отрицание. Условие $(\text{not } (a > 5))$ выполняется так же, как условие $(a \leq 5)$. Оператор **and** – это оператор конъюнкции. Итоговое условие, которое получается при объединении двух условий с помощью него, верно, только если верны оба объединённых условия. Оператор **or** – дизъюнкция, в отличие от **and**, результат верен, если верно хотя бы одно из объединяемых условий.

Эти операторы вводятся естественным образом, что позволяет читать их, не задумываясь каждый раз. Так например условный оператор **if ((a >= b) or (b > 150)) and (not (a < 0)) then ...** читается: *если a больше или равно b или b больше 150, и (при этом) не верно, что a меньше 0, то ... (выполнить следующие команды)*. Обратите внимание на расстановку скобок: если бы внутренних скобок вокруг **or**-а не было, то, так как по умолчанию больший приоритет имеет дизъюнкция (иногда её называют логическим умножением, а конъюнкцию - логическим сложением), условие читалось бы так: **if ((a >= b) or (b > 150) and (not (a < 0))) then ...** \implies *если a больше или равно b или одновременно оба следующих условия: b больше 150, и не верно, что a меньше 0, то ...*

В результате условие столкновения с платформой выглядит так:

```
if ( (x > px - 50) and (x < px + 50) and (y > image1.height - 30) ) then
begin
    отражение шарика вверх, перерисовка платформы (она может быть испорчена шариком)
end;
```

P.S. Кроме того платформа обычно делается не плоской, а выпуклой, чтобы при попадании шарика ближе к её краю он отражался с большим наклном в соответствующую сторону, что позволяет регулировать направление полёта. Рисование выпуклой платформы вместо прямоугольника не представляет существенной проблемы, и не является принципиально важным, однако написать такое отражение оказывается сложнее. Для корректного описания отражения шарика от платформы конкретной формы нужно достаточно хорошо знать геометрию, поэтому для упрощения можно рисовать платформу как угодно, например так же прямоугольную, и делать поправку к направлению в зависимости от разности $(x - x_p)$. Для этого придётся использовать введённые ранее обозначение величины скорости V и угла φ . В них отражение от вертикальной стенки выглядит не как $v_x := -v_x$, а $\varphi := \pi - \varphi$, а отражение от горизонтальной стенки: $\varphi := -\varphi$. И тогда при отражении от платформы можно делать небольшую добавку $\varphi := -\varphi + \frac{(x_p - x)}{k}$, где k определяет, насколько сильное отклонение от нормального отражения будет получаться (чем больше k , тем меньше отклонение). Разумное значение k должно быть порядка размера платформы или в несколько раз больше. В рассмотренном выше примере при размере платформы 100 разумно взять k из диапазона 100 – 500. При желании можно дать пользователю возможность регулировки этого параметра. В этой формуле принципиально, что x_p - координата именно середины платформы, в результате чего $(x_p - x)$ может быть как положительно, так и отрицательно, что даст отклонение в разные стороны. После пересчёта угла по этим формулам v_x и v_y выражаются через V и φ стандартным способом.

Столкновение с блоками

Столкновение с блоком может произойти с одной из 4 сторон либо с одного из 4 углов. При столкновении со стороной (например верхней) центр шарика находится напротив неё (для верхней - $l[i] < x < r[i]$),

а расстояние до неё меньше радиуса шарика (для верхней - $(y - t[i]) \leq r$). При столкновении с углом ни одна из координат не зажата между координатами сторон (как x в примере для верхней стороны) и при этом расстояние от угла до центра меньше радиуса (расстояние между точками (x_1, y_1) и (x_2, y_2) считается по теореме Пифагора: $l = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

Столкновение со стороной блока аналогично столкновениям со стенками, столкновение же с углом описывается не так хорошо. Точный расчёт его слишком сложен, поэтому я предлагаю две модели, которые будут давать кажущийся разумным результат. Первая - отскок назад (поворот на 180° , то есть $\varphi := \varphi + \pi$ при угловом описании столкновений или $\mathbf{vx} := -\mathbf{vx}$; $\mathbf{vy} := -\mathbf{vy}$). Его просто написать и он естественно выглядит.

Вторая - это отскок под случайным углом, кроме как внутрь блока. Такой случайный угол можно сгенерировать командами:

```
phi := random() * 3 * Pi / 2;  
phi := random() * 3 * Pi / 2 + Pi / 2;  
phi := random() * 3 * Pi / 2 + Pi;  
phi := random() * 3 * Pi / 2 - Pi / 2;
```

для левого верхнего, левого нижнего, правого нижнего и правого верхнего углов соответственно. Это на самом деле близко к правде, так как малые сдвиги начального положения сильно влияют на направление отскока при точном расчёте, так что визуально определить куда должен отлететь шарик почти невозможно. Этот способ лучше тем, что позволяет изменить направление шарика и выбить его из четырёх направлений, в которых он летает в обычных ситуациях, однако его написать несколько сложнее.

Таким образом очевидным решением является перебор циклом всех блоков в процедуре таймера (то есть при каждом сдвиге шарика) и написание для каждого восьми **if**-ов, срабатывающих при столкновении со стороной или углом, внутри которых нужно отмечать блок как разбитый, стирать его, и менять направление движения шарика. При этом в **if**-ах нужно добавлять условие, что блок, проверка столкновения с которым происходит, ещё не разбит. Считая количество разбитых блоков, можно узнать, когда игра закончится.

P.S. На самом деле можно прописать столкновения значительно более компактно, особенно если делать отражение от угла первым способом, и считать, что столкновение с блоком происходит не когда центр шарика лежит в области, показанной на рис. 3 слева (что правильно), а в области, показанной на нём справа (что на самом деле неправильно, но отличия невелико, а проверки становятся значительно проще).

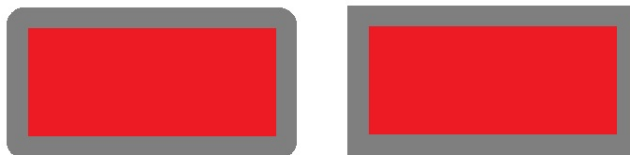


Рис. 3: Область столкновения, слева – правильная, справа – упрощённая.