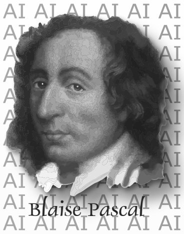
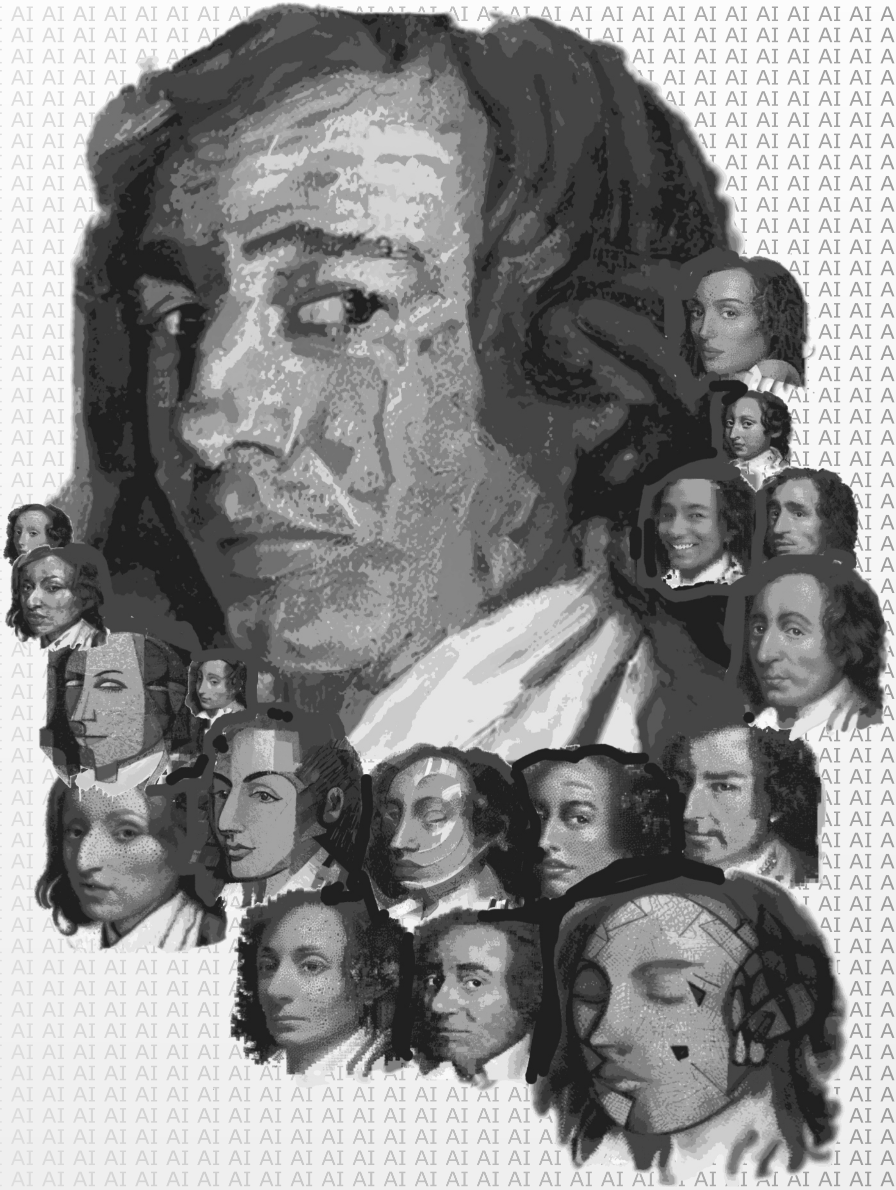
BLAISE PASCAL MAGAZINE 112

Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux



Blaise Pascal



Report Cannon ball simulation
The Castle Game Engine, the bad way to play chess:
3d physics fun using castle game engine
H-BOT, H shaped robot: a simulated robot
make and use your own robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
Executing Programs on the server in PAS2JS
long-running processes on the server

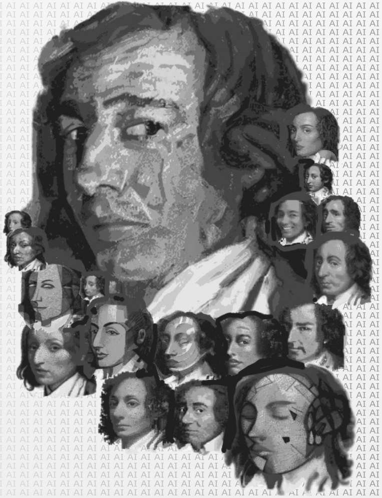
BLAISE PASCAL MAGAZINE 112

Multi platform /Object Pascal / Internet / JavaScript / Web Assembly / Pas2Js /
Databases / CSS Styles / Progressive Web App
Android / IOS / Mac / Windows & Linux


Blaise Pascal

CONTENT

ARTICLES



ADVERTISING

Watch our Blaise Pascal Artificial Gallery:
`https://www.blaisepascalmagazine.eu/pascal-portray-gallery-1/`


Niklaus Wirth

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed (left below) in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. A derivative known as Object Pascal designed for object-oriented programming was developed in 1985. The language name was chosen to honour the Mathematician, Inventor of the first calculator: Blaise Pascal (see top right).

Publisher: PRO PASCAL FOUNDATION in collaboration © Stichting Ondersteuning Programmeertaal Pascal

CONTRIBUTORS

Stephen Ball
http://delphiaball.co.uk
DelphiABall

Dmitry Boyarintsev
dmitry.living @ gmail.com

Michaël Van Canneyt
,michael @ freepascal.org

Marco Cantù
www.marcocantu.com
marco.cantu @ gmail.com

David Dirkse
www.davdata.nl
mail: David @ davdata.nl

Benno Evers
b.evers @
everscustomtechnology.nl

Bruno Fierens
www.tmssoftware.com
bruno.fierens @ tmssoftware.com

Holger Flick
holger @ flixments.com

Mattias Gärtnernc-
gaertnma@netcologne.de

Max Kleiner
www.softwareschule.ch
max @ kleiner.com

John Kuiper
john_kuiper @ kpnmail.nl

Wagner R. Landgraf
wagner @ tmssoftware.com

Vsevolod Leonov
vsevolod.leonov@mail.ru

Andrea Magni
www.andreamagni.eu andrea.
magni @ gmail.com
www.andreamagni.eu/wp

Paul Nauta PLM Solution
Architect CyberNautics
paul.nauta @ cybernautics.nl

Kim Madsen
www.component4developers.com
kbmMW

Boian Mitov
mitov @ mitov.com

Jeremy North
jeremy.north @ gmail.com

Detlef Overbeek
- Editor in Chief
www.blaisepascal.eu
editor @ blaisepascal.eu

Anton Vogelaar
ajv @ vogelaar-electronics.com

Danny Wind
dwind @ delphicompany.nl

Jos Wegman
Corrector / Analyst

Siegfried Zuhr
siegfried @ zuhr.nl

Editor - in - chief
Detlef D. Overbeek, Netherlands Tel.: Mobile: +31 (0)6 21.23.62.68
News and Press Releases email only to editor@blaisepascal.eu

Subscriptions can be taken out online at www.blaisepascal.eu or by written order, or by sending an email to office@blaisepascal.eu
Subscriptions can start at any date. All issues published in the calendar year of the subscription will be sent as well.
Subscriptions run 365 days. Subscriptions will not be prolonged without notice. Receipt of payment will be sent by email.
Subscriptions can be paid by sending the payment to: **ABN AMRO Bank Account no. 44 19 60 863** or by credit card or PayPal
Name: Pro Pascal Foundation (Stichting Ondersteuning Programeertaal Pascal)
IBAN: NL82 ABNA 0441960863 BIC ABNANL2A VAT no.: 81 42 54 147 (Stichting Ondersteuning Programmeertaal Pascal)
Subscription department Edelstenenbaan 21 / 3402 XA IJsselstein, Netherlands Mobile: + 31 (0) 6 21.23.62.68 office@blaisepascal.eu

Member of the Royal Dutch Library KB KONINKLIJKE BIBLIOTHEEK Member and donor of WIKIPEDIA

Subscriptions (2023 prices)	**Internat. excl. VAT**	**Internat. incl. 9% VAT**	**Shipment**	**TOTAL**
Printed Issue (8 per year) **±60 pages :**	**€ 200**	**€ 218**	**€ 130**	**€ 348**
Electronic Download Issue (8 per year) **±60 pages :**	**€ 64,22**	**€ 70**		

COPYRIGHT NOTICE

From your editor

Hi,
In this issue we have something very special to read and use: **Castle Game Engine**.
Michalis Kamburelis is the developer for that engine and it works under Lazarus and under Delphi as well.
This engine can let you create not only beautiful games but also 3d objects.
That is a very nice sort of development.
We are busy to integrate that into Web assembly and **PAS2JS** so we could use it eventually even on the web.
I already started to think of making a normal Desktop Application that integrates some elements of the Game Engine.
Especially for supporting the user (User-Interface) it might become very interesting.

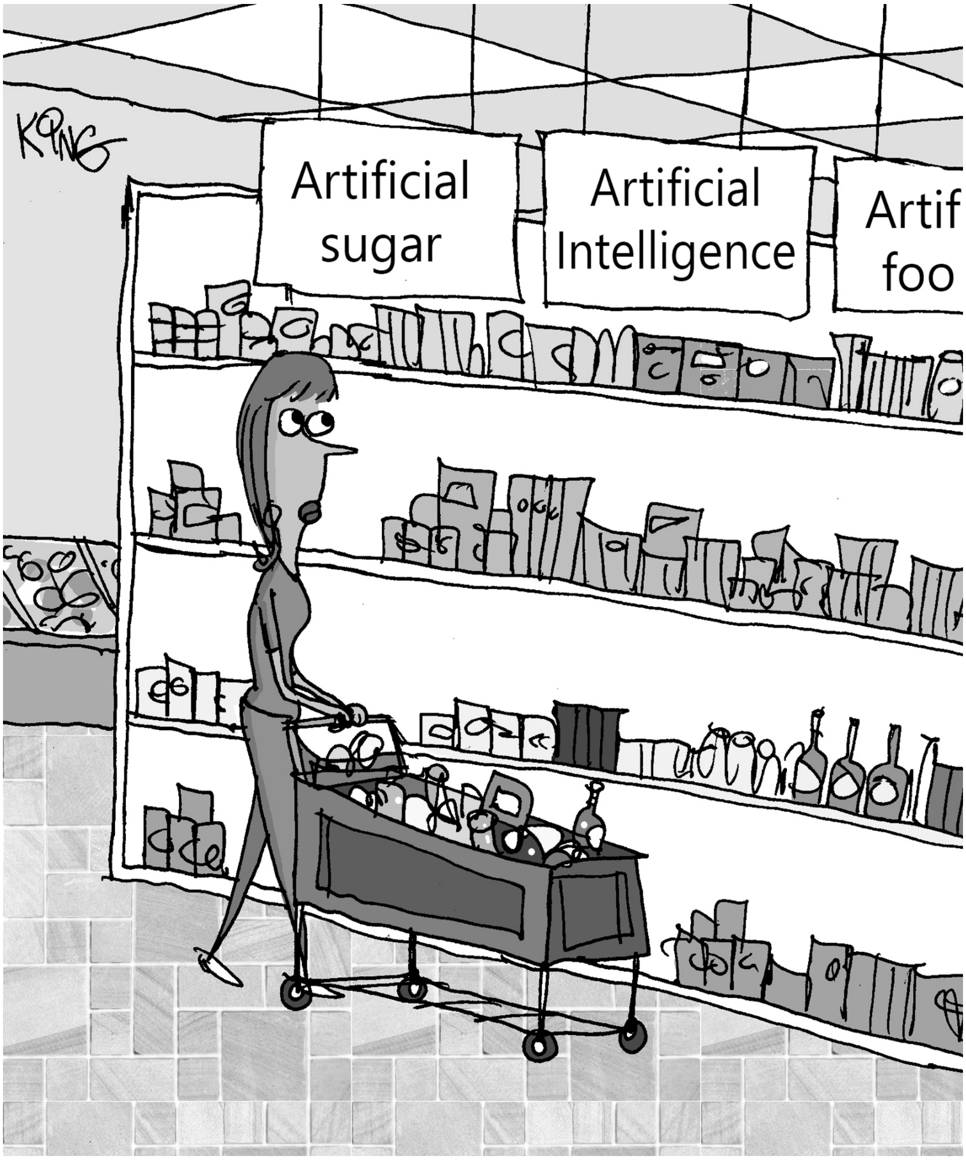
At this moment we are very deep into developing **Fresnel** (The alternative for the LCL of Lazarus) and Michael van Canneyt and Mattias Gärtner are working very hard on that.
We try to get that done by the time I will be in Backnang -Stuttgart / Heidelberg – area at the 22scd-24th of September this year.
(https://www.blaisepascalmagazine.eu/lazarus-konferenz-2023-in-backnang-22-09-2023-24-09-2023/)
Mattias tries too build something for Skia and Michael will create the extra (mouse events) for the library.
That is already an enormous step toward making Lazarus colourful.
Finally we want to do something that has no multiple OS compiler available:
Color setting as you wish – on whatever platform,
independent of the colour-scheme of the to be used OS.
Martin Friebe is continuing the very much enhanced Debugger for Lazarus. Please try, its very interesting and helpful. There are about six or seven more articles to come so it is like a course in debugging.

These new developments will be integrated into the next Lazarus Handbook as well.
Speaking about books: We will publish two new books:

1. Learning to program with FreePascal and Lazarus – it is a totally new written book with a lot of simple lessons so that you will be able to create any program you want.
2. Creating Pas2Js Apps, in Delphi and in Lazarus.

We will write about details in the next edition of Blaise Pascal Magazine.

From our technical advisor Jerry King



POCKET PACKAGE (2BOOKS)

EXCLUDING VAT AND SHIPPING

LAZARUS HANDBOOK PRICE: € 25,00

https://www.blaisepascalmagazine.eu/product-category/books/

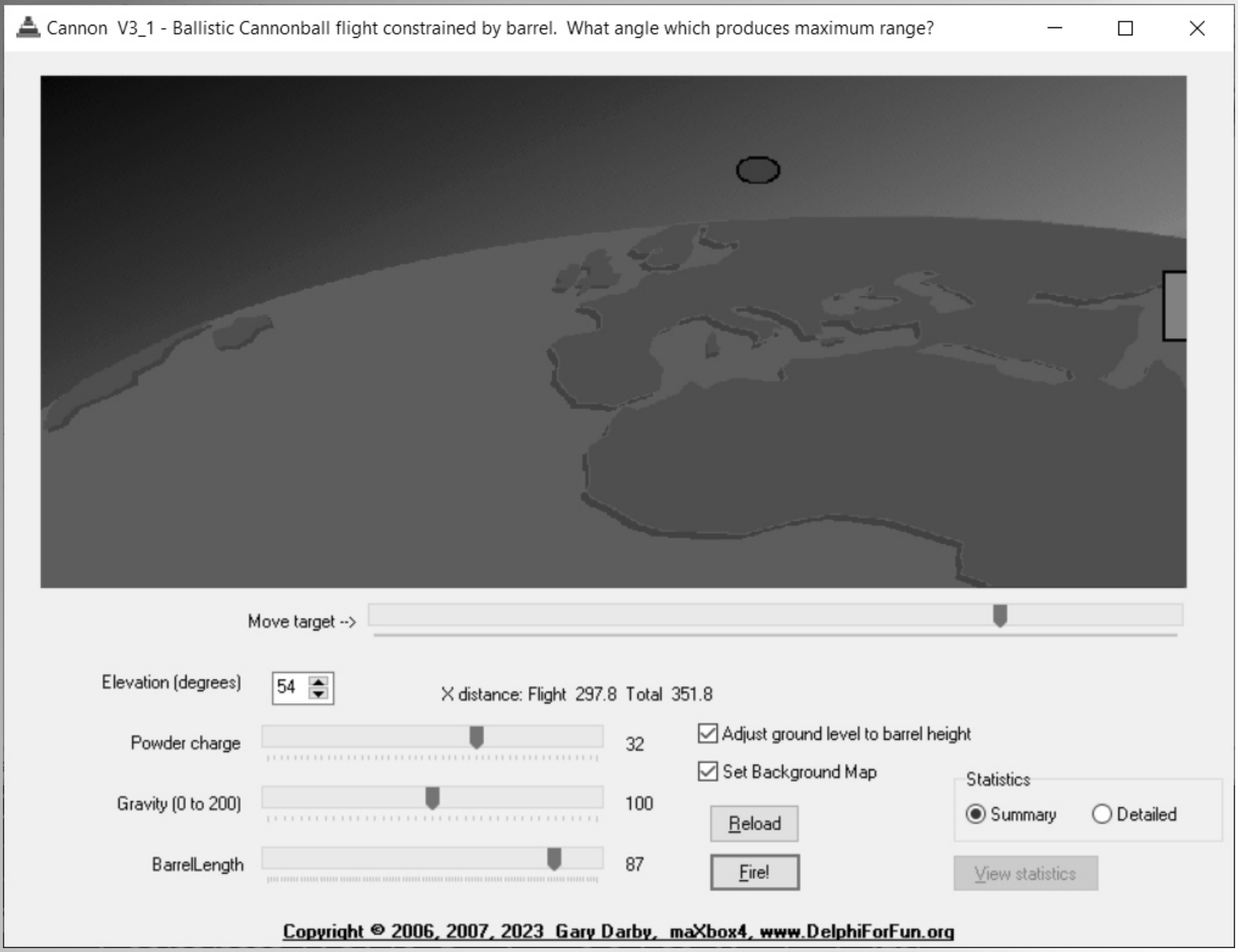
Report Cannonball Simulation

maXbox Starter 112

Starter Expert

Today we make a detour into the world of ballistics, simulation & training. One of my friends on Delphi for Fun gave me the idea to port the executable to a script for windows 11 as a preparation for the 64bitbox.

So ballistics is the study of the motion of projectiles, such as bullets, shells, and rockets, in our script we deal only with balls. It is a branch of mechanics that deals with the behavior of objects in motion. Ballistics can be divided into three main categories: internal ballistics, external ballistics, and terminal ballistics. So I translated the Delphi program with a few improvements into that script:



Cannon V3_1 - Ballistic Cannonball flight constrained by barrel. What angle which produces maximum range?

Elevation (degrees) 54

X distance: Flight 297.8 Total 351.8

Powder charge 32

Gravity (0 to 200) 100

BarrelLength 87

☑ Adjust ground level to barrel height

☑ Set Background Map

Statistics

● Summary ○ Detailed

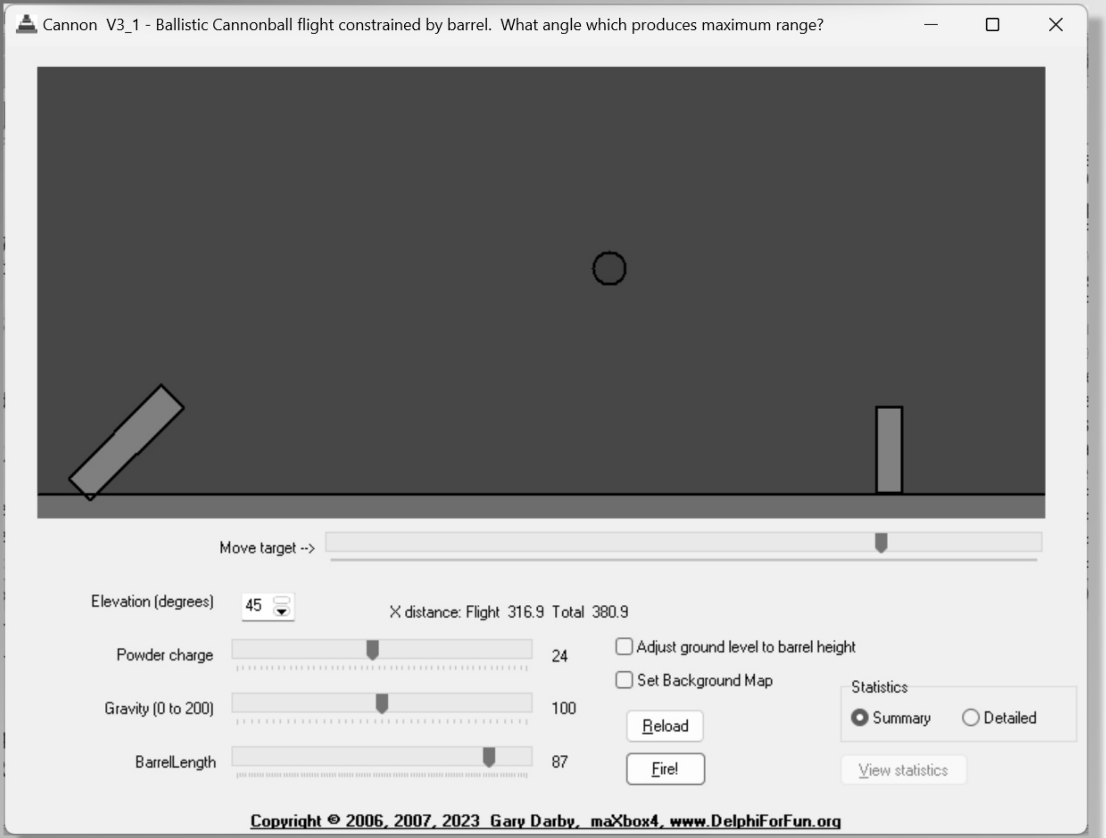
Copyright © 2006, 2007, 2023 Gary Darby, maXbox4, www.DelphiForFun.org

Of course we can simulate a cannonball using a physics simulation software or in our case integrate that model with Pascal. This simulation allows you to blast a ball out of a cannon and challenge yourself to hit a movable target. You can set parameters such as angle (elevation), initial speed (powder charge), and mass (gravity), and explore the vector representations.

`http://www.softwareschule.ch/examples/cannonball.txt`

Also an explainable statistic is part of the script, as summary or detailed (our standard case which hit the target):

```
Summary of study case
-----------------------------------------------------------------
Barrel Len 87, Angle   45.0, Initial V   24.0, gravity    1.0
Time in barrel    3.8 seconds
X distance at end of barrel   61.5
Y distance at end of barrel   61.5
Time to top of freeflight arc   15.1,   18.9 total
X distance to top of freeflight arc  226.5,  288.1 total
Height above barrel to top of freeflight arc  113.3,  174.8 total
Time to reach ground from max height   18.7,   37.6 total
X distance from top of freeflight arc to end  281.4,  569.5 total
```



Cannon V3_1 - Ballistic Cannonball flight constrained by barrel. What angle which produces maximum range?

Elevation (degrees) 45

X distance: Flight 316.9 Total 380.9

Powder charge 24

Gravity (0 to 200) 100

BarrelLength 87

☐ Adjust ground level to barrel height

☐ Set Background Map

Statistics: ● Summary ○ Detailed

Copyright © 2006, 2007, 2023 Gary Darby, maXbox4, www.DelphiForFun.org

The interesting thing is that this simulation shows how the motion of a projectile like a cannonball is fundamentally the same as the orbit of a celestial body like the moon!
The rotate and translate routines developed are used here to elevate the cannon. The ball movement loop is similar to a Bouncing Ball program with the addition of a horizontal component.
Initial velocities in the X and Y direction are proportional to the cosine and sine of the elevation angle respectively.
The barrel is a bit tricky; We do assume that the cannonball inside the barrel is "rolling up a ramp" with the component of gravity acting parallel to the barrel being the force acting to reduce the velocity of the cannonball in both x and y directions, so we keep an eye on the distance function:

`http://www.softwareschule.ch/examples/cannonball.txt`

```
function distance(p1,p2:TPoint):float;
begin
  result:= sqrt(sqr(p1.x-p2.x)+sqr(p1.y-p2.y));
end;
```

Two Procedures, `Rotate` and `Translate`, do the rotation of points. Rotation about an origin point of (0,0) is rather straightforward as we can see from the code below:

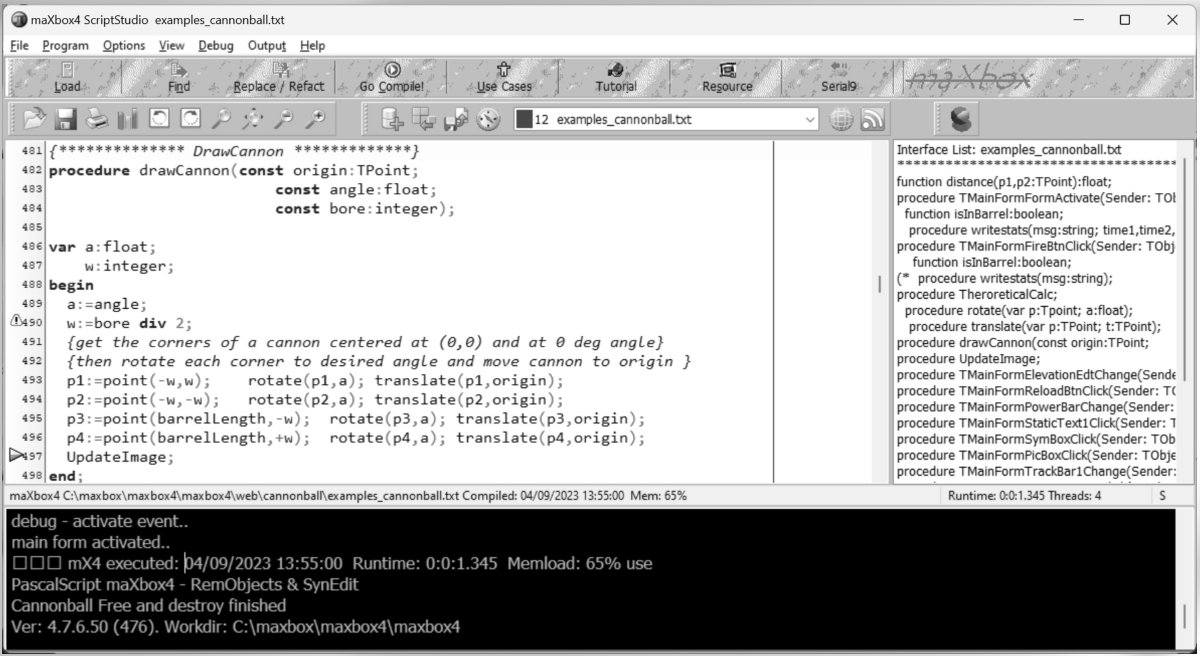
```
 procedure rotate(var p:Tpoint; a:float);
{rotate a point to angle a from horizontal}
var t:TPoint;
begin
  t:=P;
  p.x:=trunc(t.x*cos(a)-t.y*sin(a));
  p.y:=trunc(t.x*sin(a)+t.y*cos(a));
end;

procedure translate(var p:TPoint; t:TPoint);
{translate a point by t.x and t.y}
Begin
  p.x:=p.x+t.x;
  p.y:=p.y+t.y;
end;
```

Once we have the point rotated to the desired angle relative to then origin, `Translate()` can move the point by adding the new x and y origin coordinates to the x and y values of the point of type `TPoint`.

The other logic is to determine whether the cannonball has hit the target, which is movable by a track bar. "**Collision detection**" is a common (*and also complicated*) problem in most animated graphics apps.
The implementation is checking if the distance from the center of the cannonball is less than its radius from the left or top edges of the target after each move or hit.

The problem is that, for low angles, a horizontal movement may take the ball from one side of the target to the other side in one loop increment, so we never know that we went right through it!

A funny thing is the storage of cannonballs;
Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth.



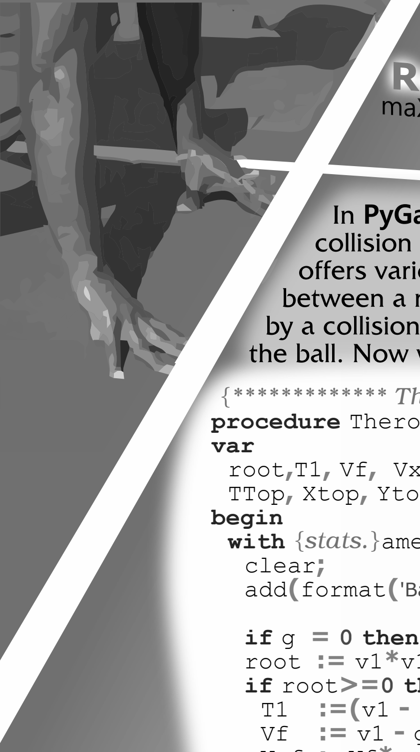
```
{************** DrawCannon *************}
procedure drawCannon(const origin:TPoint;
                           const angle:float;
                           const bore:integer);

var a:float;
    w:integer;
begin
  a:=angle;
  w:=bore div 2;
  {get the corners of a cannon centered at (0,0) and at 0 deg angle}
  {then rotate each corner to desired angle and move cannon to origin }
  p1:=point(-w,w);     rotate(p1,a); translate(p1,origin);
  p2:=point(-w,-w);    rotate(p2,a); translate(p2,origin);
  p3:=point(barrelLength,-w);  rotate(p3,a); translate(p3,origin);
  p4:=point(barrelLength,+w);  rotate(p4,a); translate(p4,origin);
  UpdateImage;
end;
```

```
Interface List: examples_cannonball.txt
*************************************
function distance(p1,p2:TPoint):float;
procedure TMainFormFormActivate(Sender: TOl
  function isInBarrel:boolean;
   procedure writestats(msg:string; time1,time2,
procedure TMainFormFireBtnClick(Sender: TObj
    function isInBarrel:boolean;
(*  procedure writestats(msg:string);
procedure TheroreticalCalc;
  procedure rotate(var p:Tpoint; a:float);
   procedure translate(var p:TPoint; t:TPoint);
procedure drawCannon(const origin:TPoint;
procedure UpdateImage;
procedure TMainFormElevationEdtChange(Sende
procedure TMainFormReloadBtnClick(Sender: T(
procedure TMainFormPowerBarChange(Sender:
procedure TMainFormStaticText1Click(Sender: T
procedure TMainFormSymBoxClick(Sender: TOb
procedure TMainFormPicBoxClick(Sender: TObje
procedure TMainFormTrackBar1Change(Sender:
```

```
maXbox4 C:\maxbox\maxbox4\maxbox4\web\cannonball\examples_cannonball.txt Compiled: 04/09/2023 13:55:00  Mem: 65%    Runtime: 0:0:1.345 Threads: 4    S
```

```
debug - activate event..
main form activated..
□□□ mX4 executed: 04/09/2023 13:55:00  Runtime: 0:0:1.345  Memload: 65% use
PascalScript maXbox4 - RemObjects & SynEdit
Cannonball Free and destroy finished
Ver: 4.7.6.50 (476). Workdir: C:\maxbox\maxbox4\maxbox4
```

http://delphiforfun.org/Programs/bouncing_ball.htm



In **PyGame** for example, collision detection is done using Rect objects. The Rect object offers various methods for detecting collisions between objects. Even the collision between a rectangular and circular object such as a paddle and a ball can be detected by a collision between two rectangular objects, the paddle and the bounding rectangle of the ball. Now we can summarize the theoretic results in a procedure of our statistic:

```
{************* TheoreticalCalc **********}
procedure TheroreticalCalc;
var
 root,T1, Vf,  Vxf, Vyf, X1,Y1  : float;
 TTop, Xtop, Ytop, Tlast, VyLast, Xlast,  floor  : float;
begin
 with {stats.}amemo1.lines do begin
  clear;
  add(format('Barrel Len %d, Angle %6.1f, Initial V %6.1f, gravity %6.1f',
                            [barrellength,180*theta/pi,v1,g]));
  if g  = 0 then g  := 0.001;
  root  := v1*v1 - 2*g*sin(theta)*Barrellength;
  if root>=0 then begin
   T1   :=(v1 - sqrt(root))/(g*sin(theta+0.001));
   Vf   := v1 - g*sin(theta)*T1;
   Vxf  :=Vf*cos(theta);
   Vyf  :=Vf*sin(theta);
   X1   :=Barrellength*cos(theta);
   Y1   :=Barrellength*sin(Theta);
   floor:=(origin.y+ballradius)-groundlevel;
   {out of barrel, Vx remains constant, Vy := Vyf- g*DeltaT}
   {Vy=0 then Vyf-g*Ttop=0 or Ttop=Vyf/g}
   Ttop:=Vyf/g;
   {x distance at top} Xtop:=Vxf*Ttop;
   {height at top = average y velocity+ time}   Ytop:=(Vyf + 0)/2*TTop;
   {Time to fall from ytop to groundlevel, descending part of projectiles path}
   {speed when ball hits ground}
   TLast:=sqrt(2*(Y1+YTop-floor)/g );
   Xlast:=Vxf*TLast;
   add(format('Time in barrel %6.1f seconds',[T1]));
   add(format('X distance at end of barrel %6.1f',[X1]));
   add(format('Y distance at end of barrel %6.1f',[Y1]));
   add(format('Time to top of freeflight arc %6.1f, %6.1f total',[Ttop,T1+Ttop]));
   add(format('X distance top of freeflight arc %6.1f, %6.1f total',[Xtop,X1+Xtop]));
   add(format('Height above barrel to top of freeflight arc %6.1f, %6.1f total',
                                          [Ytop,Y1+Ytop]));
   add(format('Time to reach ground from max height %6.1f, %6.1f total',
                                          [TLast,T1+Ttop+TLast]));
   add(format('X distance from top of freeflight arc to end %6.1f, %6.1f total',
                                          [XLast,X1+Xtop+XLast]));
  end else add('Velocity too low, cannonball does not exit barrel');
 end;
end;
```

By the way I asked **ChatGPT** how can I program cannonball in Pascal and the answer:

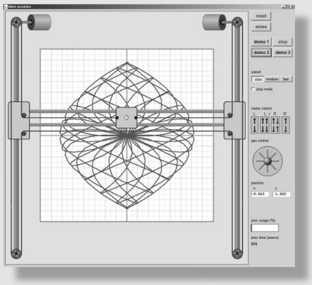
```
To program a cannonball in Pascal, you can use the following steps:

1. Define the initial position and velocity of the cannonball.
2. Calculate the acceleration of the cannonball due to gravity.
3. Update the velocity and position of the cannonball using the calculated acceleration.
4. Repeat step 3 until the cannonball collides with an object or reaches a certain height.
```

In this example code snippet, `CircleRectCollision()` is a custom function that detects collision between a circle and a rectangle. You can modify this function to suit your needs; the main part of the script has only 4 procedures:

```
processmessagesOFF;  loadStatForm();  loadmainForm();  UpdateImage();
```

https://en.wikipedia.org/wiki/Ballistics **http://www.softwareschule.ch/examples/cannonball64.txt**

H-BOT, H SHAPED ROBOT:
A SIMULATED ROBOT



INTRODUCTION:

The Picture below shows a **HBot** (*H shaped robot*).
These robots are widely used in gantries,(*A Gantry Robot is an automated industrial system that can also be referred to as a Cartesian Robot or a Linear Robot.*) manufacturing such as SMD pick-and-place, packaging and laser cutting. But also unexpected applications were found: T-shirt folding, chess playing, tattoo machine. Advantages of this robot type are cost effectiveness, portability and ease of control.

Figure 1: HBot

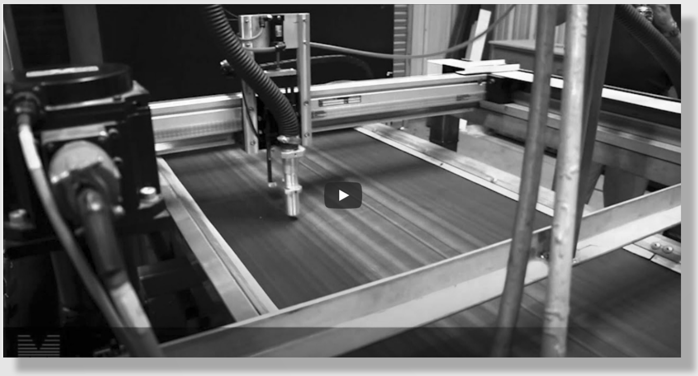
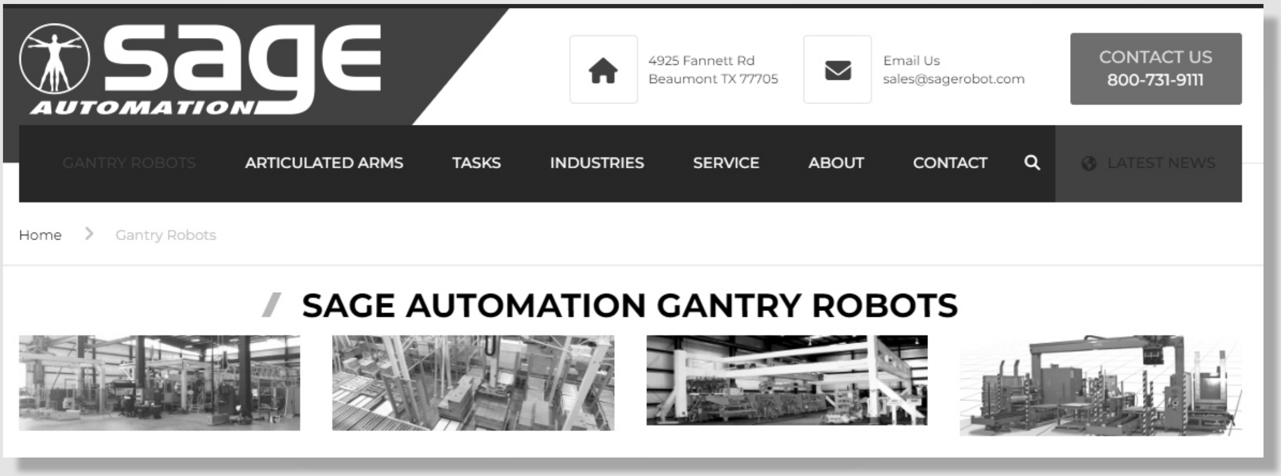


Figure 2: The video cover

Please take a look at the Youtube video (*see figure 2*)
`https://www.youtube.com/watch?app=desktop&v=NgGiu_0x7tg`



4925 Fannett Rd
Beaumont TX 77705

Email Us
sales@sagerobot.com

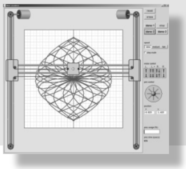
CONTACT US
800-731-9111

Home > Gantry Robots

SAGE AUTOMATION GANTRY ROBOTS

Figure 3: a very good example website

Here you can get more information about the H-shaped Robots
`https://www.sagerobot.com/gantry-robots/`



Here is a video about a totally different application:
`https://www.youtube.com/watch?v=Ztm-PrCzxos`

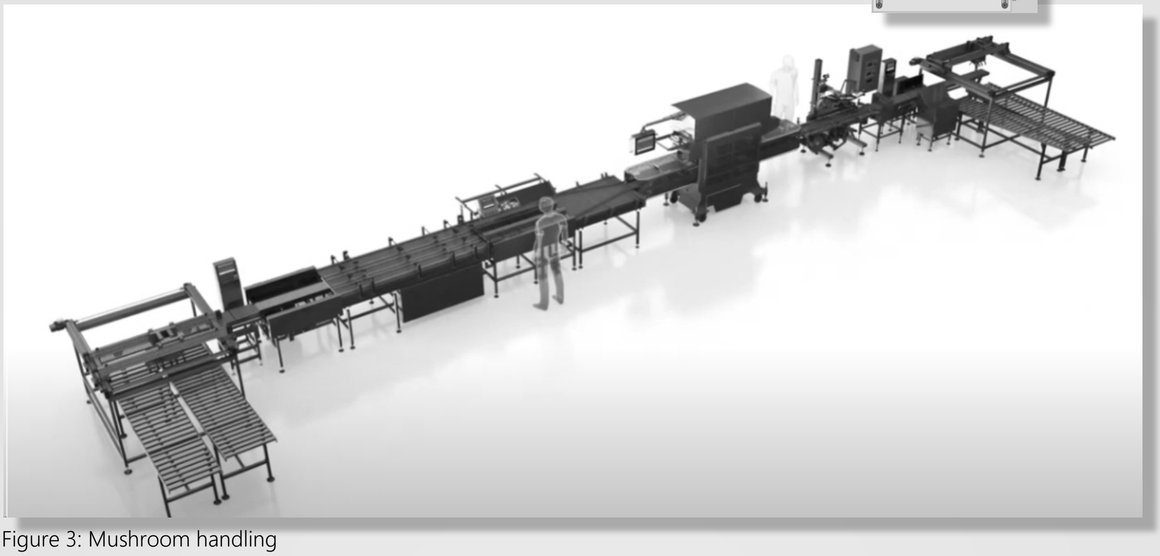


Figure 3: Mushroom handling

Pictured below (*reduced*) is the simulated robot, implemented as a plotter. At the left and right top are two fixed motors operating independently and driving a single belt. As the motors rotate clockwise or counter clockwise, the pen moves in a horizontal , diagonal or vertical direction.

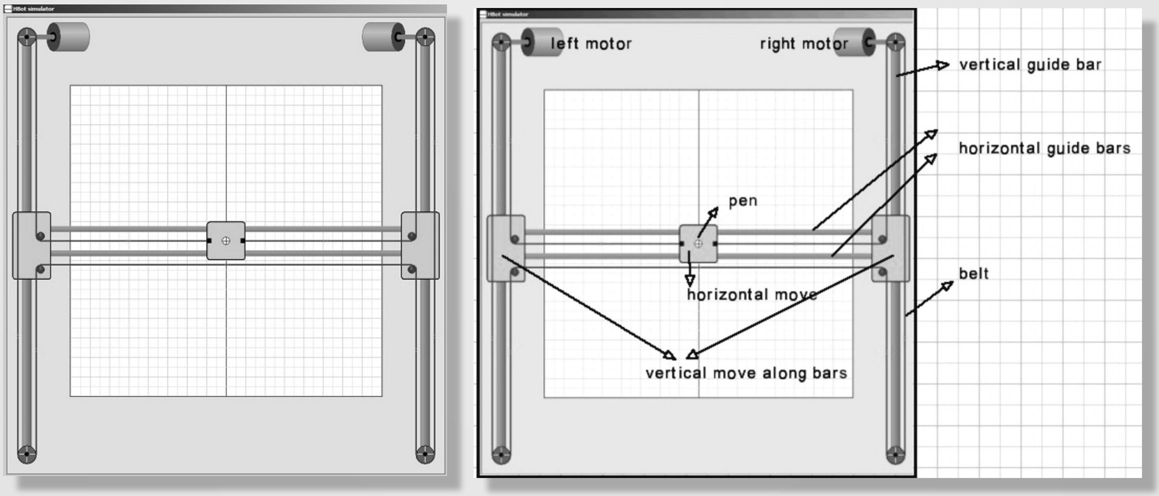
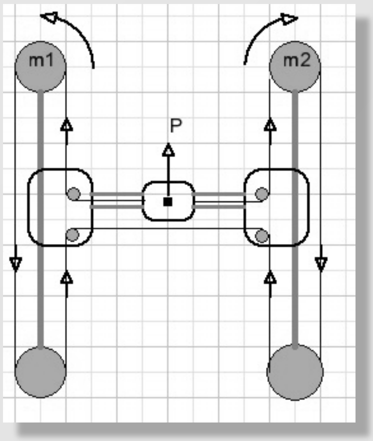
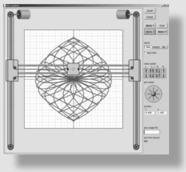


Figure 4: Motors and guide bar



There is one single belt which ends are attached to the pen.

Motor 1 moves counter clockwise,
motor 2 moves clockwise, the same distance.
Pen P moves upward.



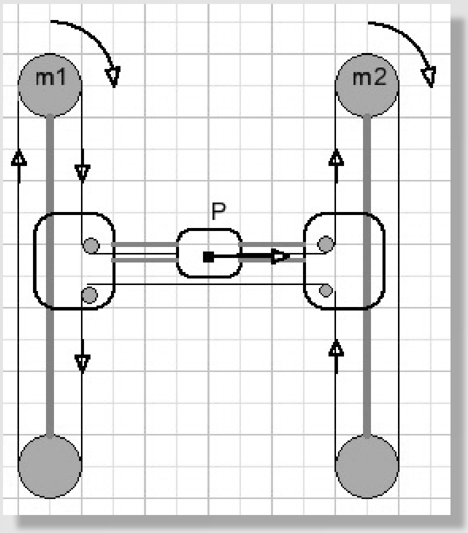


Figure 5: Horizontal Pen movements

Horizontal pen movement.

Motor 1 moves clockwise,
motor 2 also moves clockwise, the same amount.
PenP moves horizontally to the right.
Diagonal pen movement
Is the result of both vertical and horizontal movement.
For pen
Up = Left motor CCW d ; Right motor CW d (distance)
Right = Left motor CW d; Right motor CW d
----

add actions
Diagonal UP = Right motor CW 2d
Diagonal right up movement by distance d is the result of a 2d distance turn by motor 2.
The motor 1 movements cancel each other.
Summarizing:

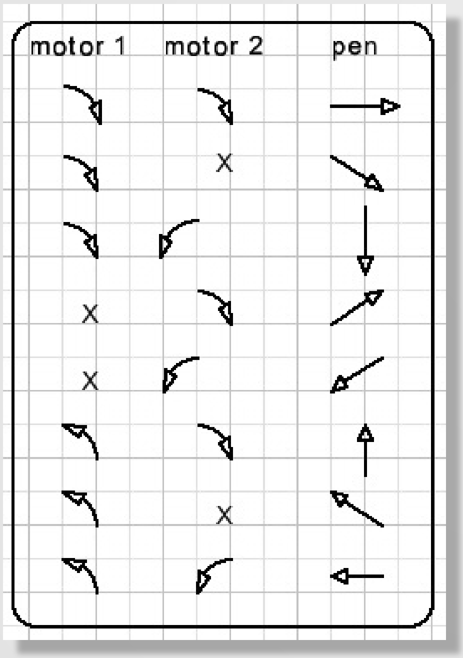
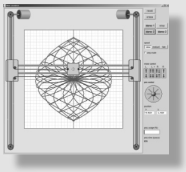


Figure 6: Actions

For diagonal movement only one motor moves.
The belt movement is doubled:
it is the sum of horizontal and vertical movement.



SIMULATOR USAGE, BUTTONS AND INDICATORS

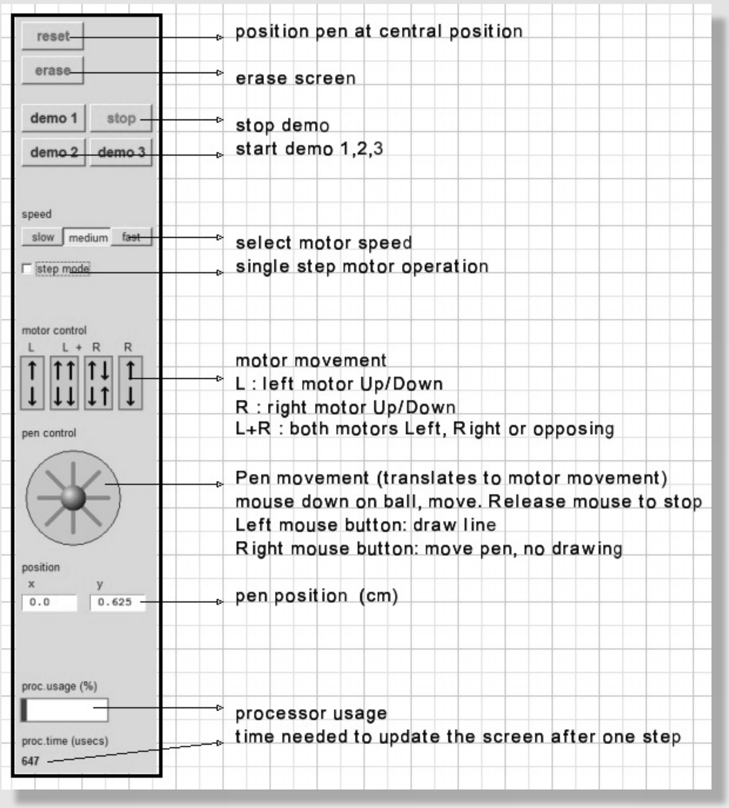


Figure 7: The program use

AREAS AND BITMAPS

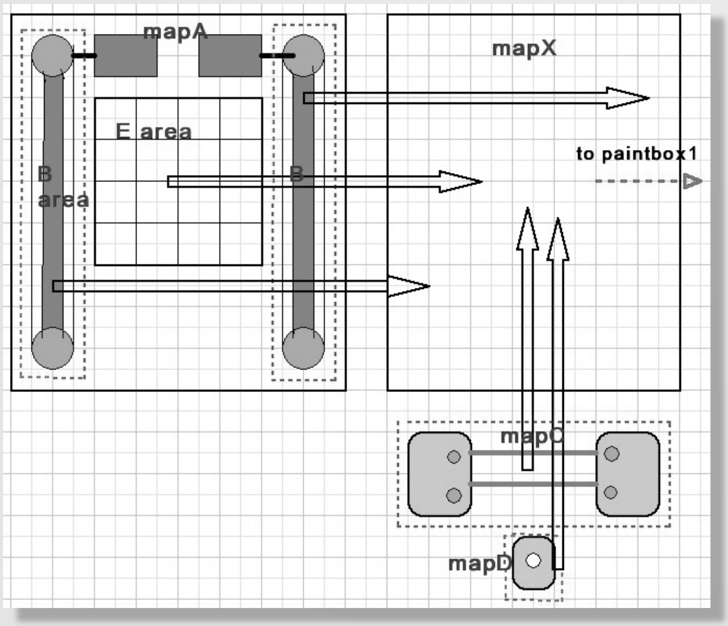


Figure 8: The mapping

THE SIMULATOR PROGRAM

General considerations

- Movement must be realistic: smooth without flickering
- Moving parts must be visible as such
- Buttons must show the principle of operation

The **HBot** is pictured in a paintbox on the main form. However, this is not sufficient because erasing the picture before drawing the updated situation causes irritant flickering.
This is avoided by assembling the HBot in a bitmap (mapX) and using copyrects to transfer changed areas of mapX to paintbox1
.
Some areas of the screen are painted once and do not change.

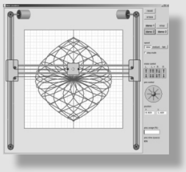
This background is painted in mapA.
Parts of A are transferred to mapX. Also the pen draws in mapA.
Moving wheels have attached spokes or a line mark to indicate movement.
Purple dots are painted on the belt to show movement.

TIMING

Updating the screen after a (one pixel) move takes some time.
This time is displayed.
After updating the screen the processor has to wait for a certain time for the required speed.
A blue bar displays the percentage of time needed for a one pixel update.

Typical time is less than 800 microseconds.

MapA : bitmap with motors and B areas with vertical guides, wheels and belt.
Area E is for drawing and shows a coordinate system.
Map A is copied to mapX
Map C: bitmap holding horizontal guide bars and wheels. Is copied to mapX.
Map D : pen holder. Is copied to mapX.
MapX is (partial) copied to paintbox1 to become visible after adding spokes to the wheels and adding dots to the belt.



Delphi code For details, please refer to the source code.
Units

- `Unit1 : simulator control, constants, variables, painting procedures.`
- `Unit2 : TDav7ELbox` is a `Tpaintbox` with enter leave events added and is used for motor control buttons.
- `Timer_unit:` code to turn CPU ticks into a microseconds clock for speed control.
- `Demo_unit`: supplies 3 demos to illustrate HBot operation.

Unit1 procedures and functions
`PaintmapA, paintmapC, paintmapD, paintLeftMotor, paintRightMotor` : self explanatory.
`PaintB` : paint area B on mapA.
`PaintE` : paint coordinate system on mapA.

```
procedure paintchain(mp: Tbitmap;x1,y1,x2,y2 : word);
// paint belt on bitmap mp (x1,y1) (x2,y2)
```

`XpartialToBox;` *//area of mapC, B areas paintbox, these are the updated rectangles of mapX*

`procAWheelmovement;` *//place spokes on wheels in B areas.*

`procCwheelmovement;` *//add spoke to small wheels in C area.*

`procBeltMovement;` *// add purple dots on belt*
For the last three procedures, the place of the spokes or dots is calculated from the position of the pen.
(0,0) is the center of area E.
The coordinates on E are (-320,-320) left top to (320,320) right bottom.
The procBeltMovement code is lengthy because the belt is divided into horizontal and vertical stretches and also the arcs around the wheels.

MOVING THE PEN.
This simulator project is for educational purposes.
For that reason there are two ways to move the pen:

1. By controlling the motors and observe pen motion.
2. By controlling the pen and observing the motors.

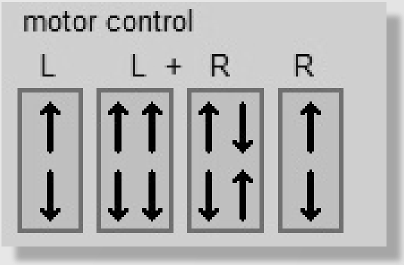


Figure 9: The motor control

Picture Right shows four paintboxes with added on-enter and on-leave events. Code is provided by unit2. These paintboxes are created at run time.

A left mouse button down on the top half of the L button moves the left top motor counterclockwise. The bottom half causes clockwise motion. These actions reverse for the right mouse button.
The R paintbox operates in a similar way, now causing clockwise motion for a mouse button down on the top part.
The L+R buttons activate both motors moving either in the same or in opposite directions.

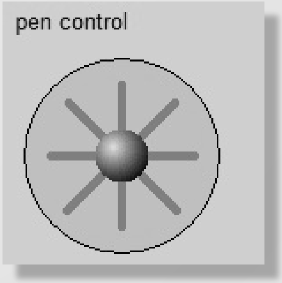
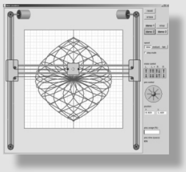


Figure 10: The pen control

TO MOVE THE PEN WITHOUT DRAWING:
Right mouse button down on the sphere, move in chosen direction. Release mouse button if pen reaches position.
Use left mouse button to move the pen for drawing.
A mouse down event on the sphere causes the movebusy flag to set.
The movebusy flag enables mouse move events.
The pen control paintbox is divided in cells, the cell number is translated to the direction code (xdircode).



Right pictured are the paintbox cells and right the direction code is displayed.
`Procedure procXpainting`
is called which sets the `moveflag` and continuously calls `procedure procmove` as long as the `moveflag` is set.
`Procmove` takes care of the speed. For single motor operations, the time out period is doubled because the motor has to turn twice the distance moving the pen in a diagonal way.
`Procmove` calls procedure `moveControl` to calculate the new pen position (`penPosX,penPosY`) , update `mapX` and copy parts of `mapX` to `paintbox1` on `form1`.
If outer bounds of the E area are reached `procMove` clears `moveFlag` and pen motion stops.

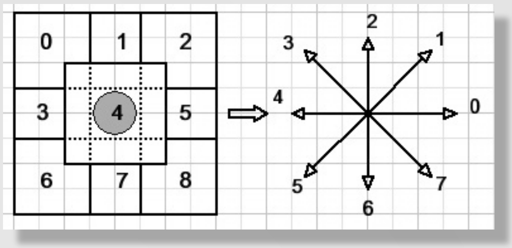


Figure 11: The paintbox cells

THE DEMO UNIT

Three demos are provided. The demo buttons have tags 1..3 and share the OnClick event. A demo button click calls `procedure startdemo(demoNr)`.

DEMO 1:
Plots the parameter function

```
X = 150(sin(8t)+sin(t))
Y = 150(cos(8t)+cos(t))
```

Where t runs from 0…2*pi in 500 steps.
For each value of t, x and y are calculated and `procedure movetoXY` is called to draw a line to the new `(x,y)` position.

DEMO 2:
Is similar to demo1 but the function is

```
X = 500(sin(9t)*cos(9t)*sin(7t))
Y = 275(sin(9t)*cos(7t))
Procedure movetoXY(x,y)
```

This procedure subsequently calls `procedure procmove( direction)` to reach pen position `(x,y);`
This is done by calculating values `Xstep` and `Ystep` first and then incrementing x (by Xstep) and y (by Ystep);
Difference must be made between horizontal and vertical line orientation.

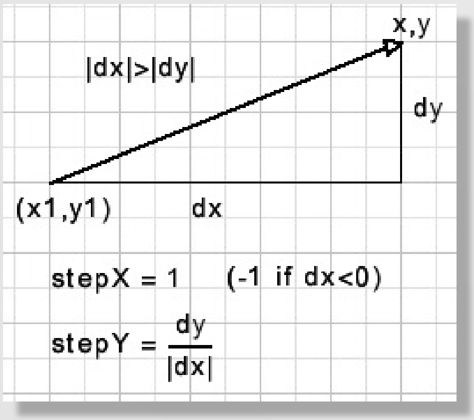


Figure 12: horizontal orientation

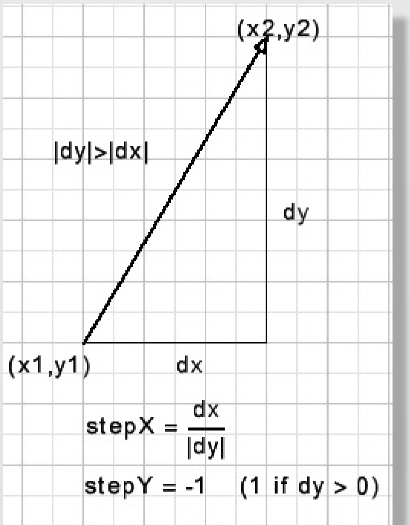
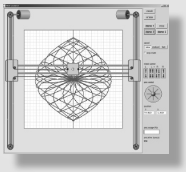


Figure 13: vertical orientation



DEMO 3:

This demo **plots some lines of text.**
Text is preset in array `demotext` together with the coordinates of the first character, the font height and of course the character string.
`Procedure startdemo(3)` calls `procedure painttextline(line nr)` for each line of text .
`Painttextline` calls procedure `drawdemochar(x,y)` for each character of the line.
`Drawdemochar` finally calls `movotoXY(x,y)` and `procmove(direction)`.
Remains to explain how the parameters for `procedures movetoXY (  )` and `procmove(   )` are calculated.
A bitmap called `scanmap (width=60, height=60 pixels )` is erased for each character with a black background. Then the character is painted in `scanmap` with `pencolor red` and `background white`. The black color indicates the boundaries of the character.

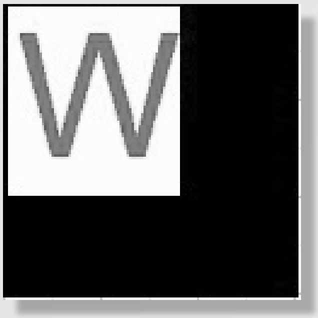


Figure 14: drawing a character

SCANMAP

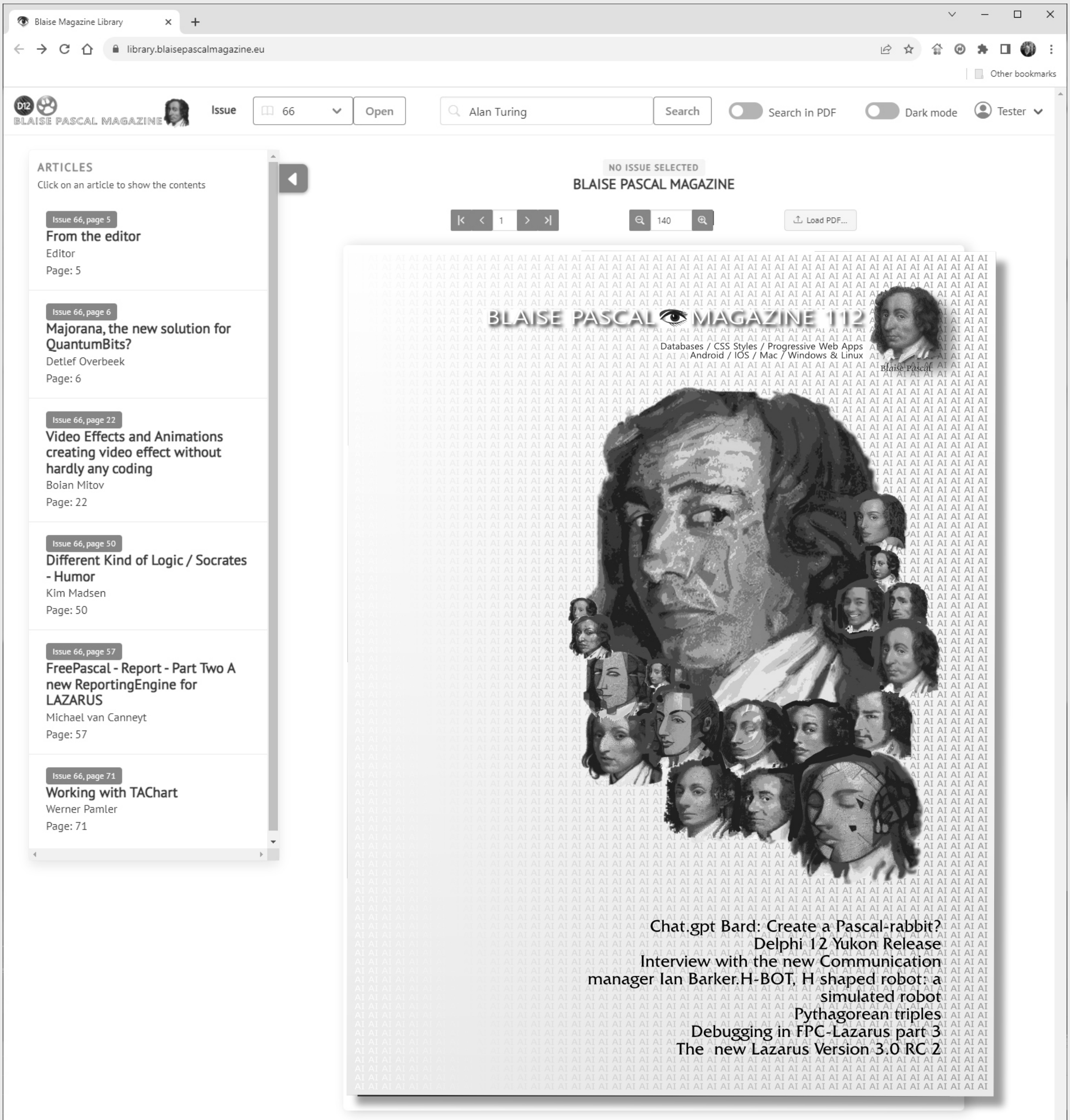
`Drawdemochar` has variables `scanX, scanY` and `x, y` which point to pixel positions of scanmap.
First `scanX` and `scanY` are set to 0;
Then function `scanChar(var scanX,scanY) : Boolean;` is called.
This function scans the `scanmap` (*left to right, top to bottom*) to find a red pixel and return true after that red pixel is set to white.
`ScanX,scanY` point to the first red pixel found.

Next `drawdemochar` lowers the pen to set a dot. `x` is set to `scanX` , `y` is set to `scanY` and `function subscan(var x,y; var dir) : Boolean` is called.
This function searches for neighbour red pixels of x,y.
If found, true is returned together with the direction code needed for the call to `procedure procmove( )` to step the pen.
Also `x,y` coordinates are updated to reflect the last red pixel position.
If `subscan` returns false, `scanChar` is called again to continue the search for remaining red dots, starting at `scanX,scanY`.
The roman font is used to paint the characters in `scanmap`.
The `GoDemoFlag` must be true for `textdrawing` to continue.
A click on the stop button clears the flag, ending the demo.

* * *

THE NEW SUBSCRIPTION MODEL OF BLAISE PASCAL MAGAZINE

1. SUBSCRIPTION: PER YEAR - NOTHING CHANGES ISSUES STARTING AT THE LATEST ISSUE AVAILABLE +1 YEAR / CODE INCLUDED € 70,00 FOR ALL COUNTRIES
2. INTERNET (LIBRARY) USE FOR ALL MAGAZINES FROM 1- THE LATEST ISSUE € 50,00 FOR ALL COUNTRIES
3 LIB-STICK USB-CARD: ALL ISSUES / CODE INCLUDED. SAME INTERFACE AS THE INTERNET LIBRARY.€ 120,00 FOR ALL COUNTRIES



Issue: 66

Search: Alan Turing

ARTICLES

Click on an article to show the contents

Issue 66, page 5
From the editor
Editor
Page: 5

Issue 66, page 6
Majorana, the new solution for QuantumBits?
Detlef Overbeek
Page: 6

Issue 66, page 22
Video Effects and Animations creating video effect without hardly any coding
Boian Mitov
Page: 22

Issue 66, page 50
Different Kind of Logic / Socrates - Humor
Kim Madsen
Page: 50

Issue 66, page 57
FreePascal - Report - Part Two A new ReportingEngine for LAZARUS
Michael van Canneyt
Page: 57

Issue 66, page 71
Working with TAChart
Werner Pamler
Page: 71

NO ISSUE SELECTED

BLAISE PASCAL MAGAZINE

BLAISE PASCAL MAGAZINE 112

Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

Blaise Pascal

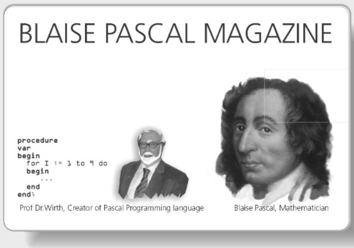
Chat.gpt Bard: Create a Pascal-rabbit?
Delphi 12 Yukon Release
Interview with the new Communication manager Ian Barker.H-BOT, H shaped robot: a simulated robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
The new Lazarus Version 3.0 RC 2

https://www.blaisepascalmagazine.eu/product-overview/

USE WHERE EVER THE INTERNET IS AVAILABLE

LIB-STICK ON USB CREDIT CARD
BLAISE PASCAL MAGAZINE

LIB-STICK USB-CARD: ALL ISSUES / CODE INCLUDED. SAME INTERFACE AS THE INTERNET LIBRARY € 120,00

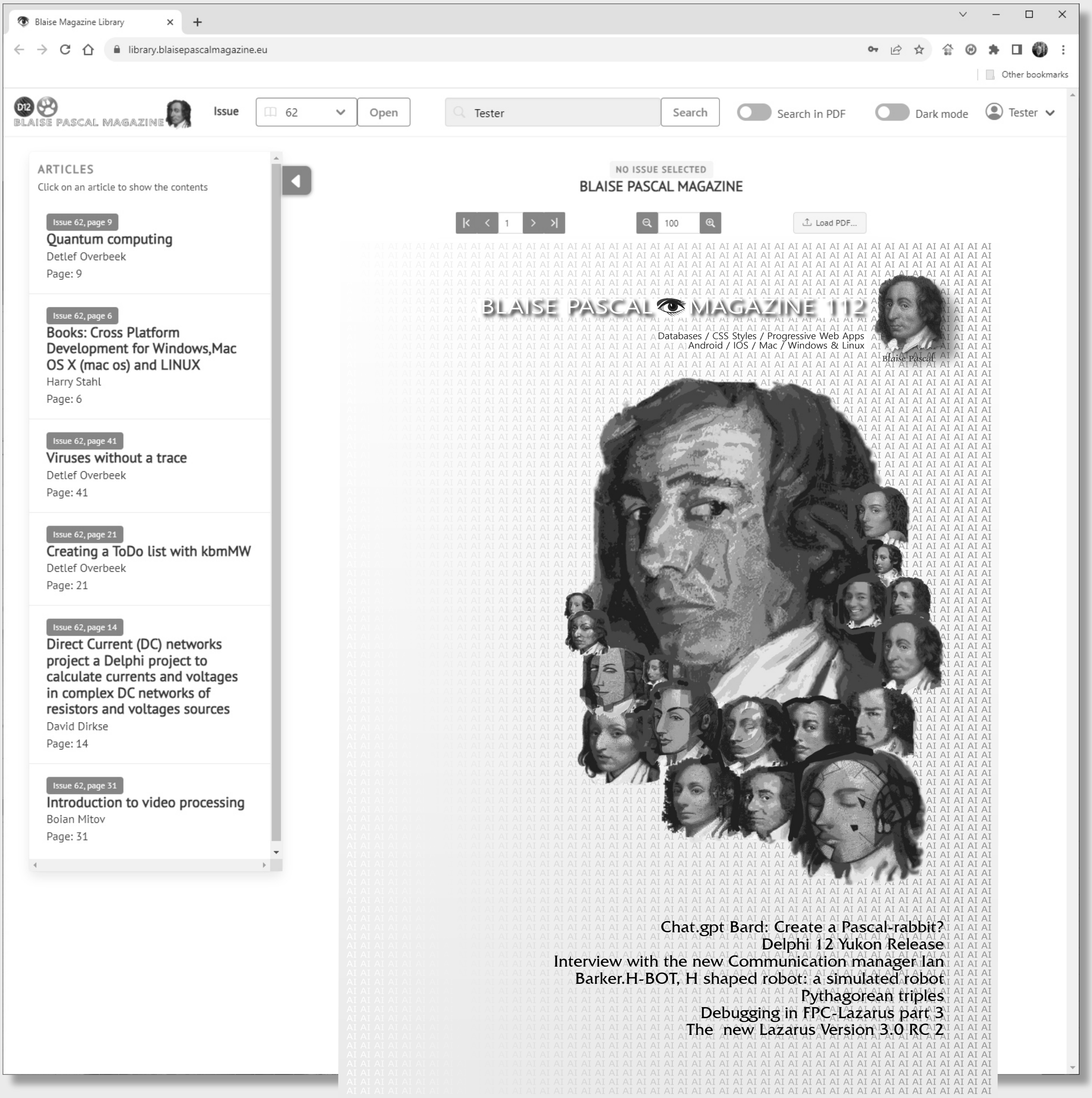


```
procedure
var
begin
  for I := 1 to 9 do
  begin
     ...
  end
end;
```


Editor in Chief: Detlef Overbeek
Edelstenenbaan 21 3402 XA
IJsselstein Netherlands

editor@blaisepascalmagazine.eu
https://www.blaisepascalmagazine.eu

```
procedure
var
begin
  for I := 1 to 9 do
  begin
     ...
  end
end;
```



Issue: 62

Search: Tester

ARTICLES

Click on an article to show the contents

Issue 62, page 9
Quantum computing
Detlef Overbeek
Page: 9

Issue 62, page 6
Books: Cross Platform Development for Windows,Mac OS X (mac os) and LINUX
Harry Stahl
Page: 6

Issue 62, page 41
Viruses without a trace
Detlef Overbeek
Page: 41

Issue 62, page 21
Creating a ToDo list with kbmMW
Detlef Overbeek
Page: 21

Issue 62, page 14
Direct Current (DC) networks project a Delphi project to calculate currents and voltages in complex DC networks of resistors and voltages sources
David Dirkse
Page: 14

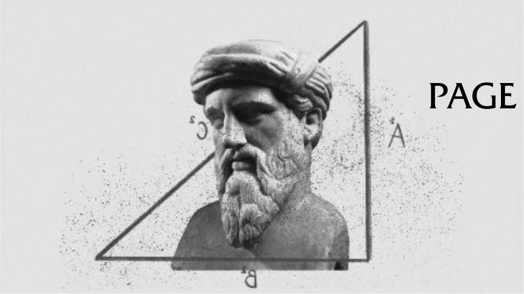
Issue 62, page 31
Introduction to video processing
Boian Mitov
Page: 31

NO ISSUE SELECTED
BLAISE PASCAL MAGAZINE

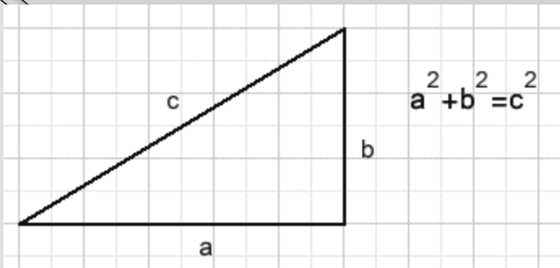
Page: 1 — Zoom: 100

BLAISE PASCAL MAGAZINE 112
Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux
Blaise Pascal

Chat.gpt Bard: Create a Pascal-rabbit?
Delphi 12 Yukon Release
Interview with the new Communication manager Ian Barker.H-BOT, H shaped robot: a simulated robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
The new Lazarus Version 3.0 RC 2



7 Starter Expert D11



INTRODUCTION

In secondary school students are confronted with the theorem of **Pythagoras**:
In a right angled triangle, the square of the hypothenuse c equals the sum of squares of the right angled sides a and b.
Calculations with above formula usually result in roots, numbers that only may be approximated.
Some values of a and b however result in an integer value of c, such as

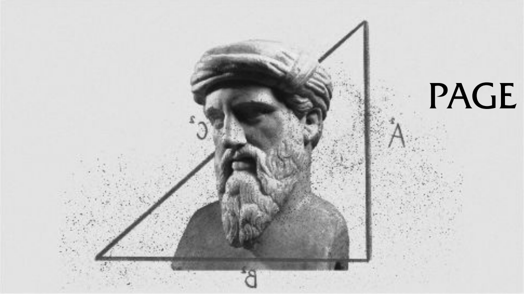
- 3,4 c=5
- 5,12 c=13

(3,4,5) and (5,12,13) are called Pythagorean triples.
The question arises: are there more triples?
This Delphi project was written to find all triples below 1000.

ANALOGOUS TRIPLES

(3,4,5) being a triple involves that also (6,8,10), (9,12,15) .. are triples of similar triangles.
A simple criterion `GCD(a,b) = 1` eliminates these analogous triples

```
function GCD(a,b : word) : word; //greatest common divisor
var h : word;
begin
 repeat
  if a < b then
   begin
    h := a;
    a := b;
    b := h;
   end;
  a := a mod b;
 until a = 0;
 result := b;
end;
```



This function uses the **Euclidean lemma**:
GCD(a,b) = GCD(a mod b,b).

Example: `GCD(77,21) = GCD(14,21)=GCD(21,14)=GCD(7,14)=GCD(14,7)=GCD(0,7)=7`

Time measurement.
A microseconds timer component is added to the project.
To find the real processing time without the burden of reporting (`memo1.lines.add( string)`)
detected triples are first stored in array `ptriples[ ]`

```
type TP3 = record
       a,b,c : word;
      end;
...
var p3nr : byte; //sequence number of triple
  ptriples : array[1..200] of TP3;
```

The project presented here has three selectable procedures to find Pythagorean triples.

METHOD NR 1.

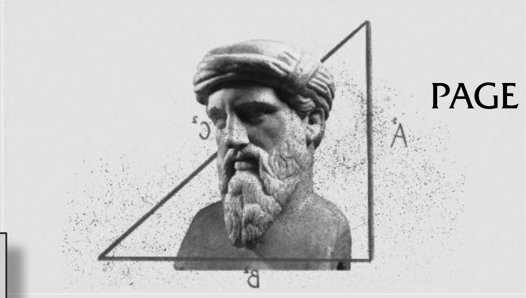
Uses no floating point operations and a preset table of squares.

```
Var   squares : array[1..1500] of dword;
...
procedure presetsquares;
var i : word;
begin
 for i := 1 to 1500 do squares[i] := i*i;
end;
```

SUMMARY:

```
 Var c2 : dword;
...
{a and b are incrementing variables in nested repeat..until loops}
C2 := squares[a] + squares[b];
C  := b;
{a third nested repeat..until loop increments c}
If  c2 =  squares[c] then  ....   //report a,b,c  as new triple
```

This is the slowest method. Processing time is over 100 milliseconds.



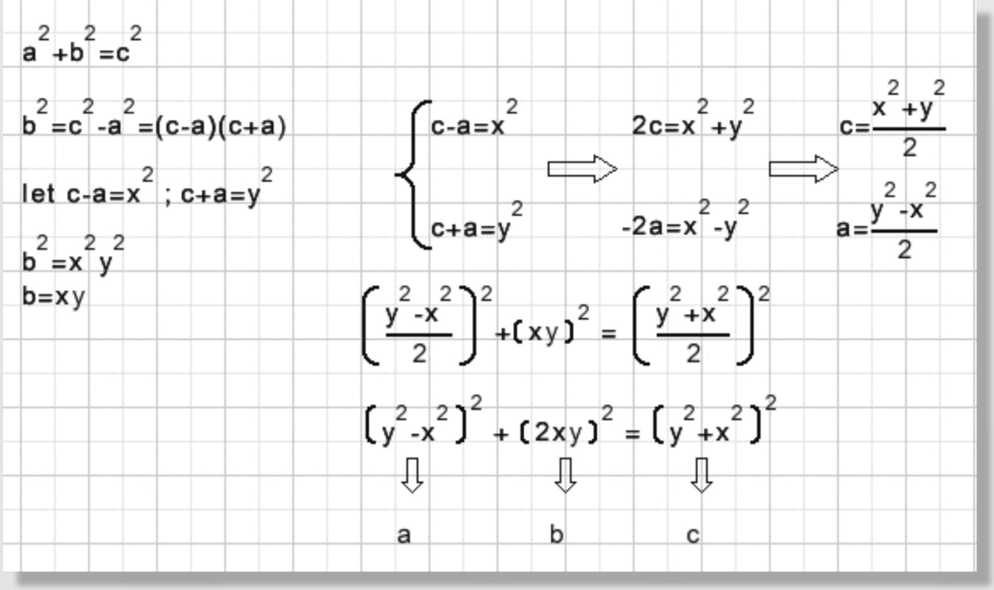
METHOD2

```
procedure p3method2;
var a,b : word;
   a2,b2 : dword;
   c : single;
begin
 form1.proctimer.start;
 for a := 1 to 998 do
  begin
   a2 := a*a;
   for b := a+1 to 999 do
    begin
     b2 := b*b;
     c := sqrt(a2 + b2);
     if frac(c) = 0 then
      if GCD(a,b) = 1 then addtriple(a,b,round(c));
    end;
  end;
 form1.proctimer.stop;
end;
```

This method has two nested for loops to increment a and b.
It uses floating point operations sqrt( ) and `frac( )` to calculate the square root and extract the fraction.
If `frac(V) = 0`, the value v is integer.
Processing time is around 40 milliseconds.

METHOD3

This method does not search for triples but uses formulas that yield triples.
Below is the theory:



$$a^2+b^2=c^2$$

$$b^2=c^2-a^2=(c-a)(c+a)$$

$$\text{let } c-a=x^2 \;;\; c+a=y^2$$

$$b^2=x^2y^2$$

$$b=xy$$

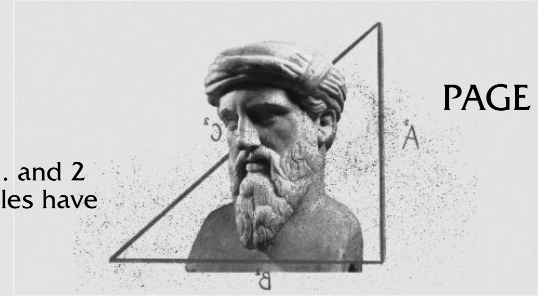
$$\left\{\begin{array}{l} c-a=x^2 \\ c+a=y^2 \end{array}\right. \Rightarrow \begin{array}{l} 2c=x^2+y^2 \\ -2a=x^2-y^2 \end{array} \Rightarrow \begin{array}{l} c=\dfrac{x^2+y^2}{2} \\ a=\dfrac{y^2-x^2}{2} \end{array}$$

$$\left(\frac{y^2-x^2}{2}\right)^2+(xy)^2=\left(\frac{y^2+x^2}{2}\right)^2$$

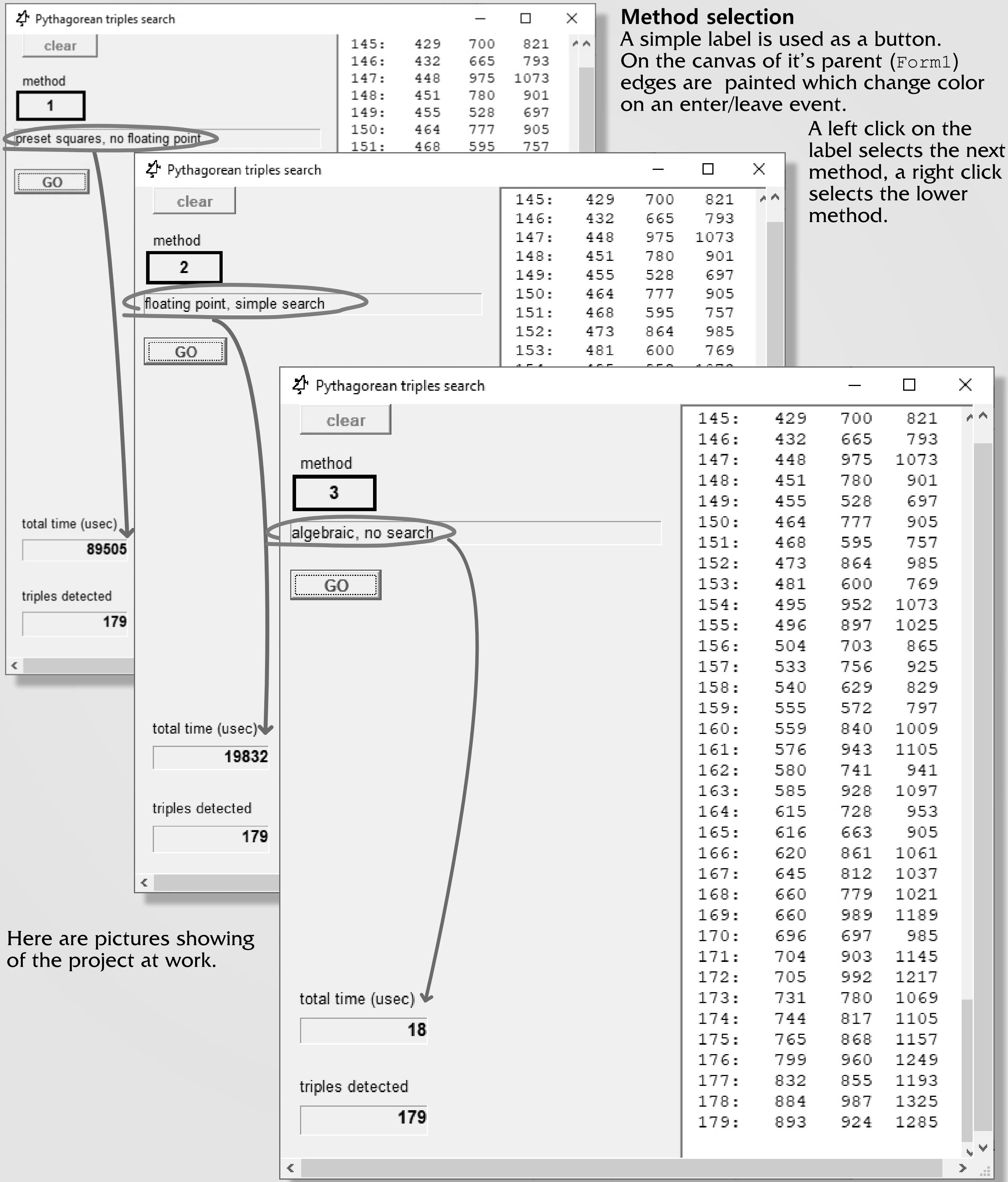
$$\underset{\substack{\Downarrow \\ a}}{(y^2-x^2)^2}+\underset{\substack{\Downarrow \\ b}}{(2xy)^2}=\underset{\substack{\Downarrow \\ c}}{(y^2+x^2)^2}$$

a,b,c are replaced by x,y.

PYTHAGOREAN TRIPLES



Each combination where x<> y generates a triple.
Processing time is 85 microseconds. Unlike methods 1. and 2 were a and b were systematically incremented, the triples have to be sorted to obtain the same result sequence.
A simple exchange sort procedure is used.
Sorting time is not measured.



method 1

preset squares, no floating point

total time (usec) 89505

triples detected 179

```
145:   429   700   821
146:   432   665   793
147:   448   975  1073
148:   451   780   901
149:   455   528   697
150:   464   777   905
151:   468   595   757
```

method 2

floating point, simple search

total time (usec) 19832

triples detected 179

```
145:   429   700   821
146:   432   665   793
147:   448   975  1073
148:   451   780   901
149:   455   528   697
150:   464   777   905
151:   468   595   757
152:   473   864   985
153:   481   600   769
```

method 3

algebraic, no search

total time (usec) 18

triples detected 179

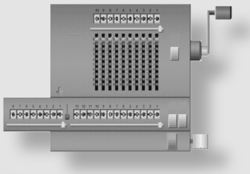
```
145:   429   700   821
146:   432   665   793
147:   448   975  1073
148:   451   780   901
149:   455   528   697
150:   464   777   905
151:   468   595   757
152:   473   864   985
153:   481   600   769
154:   495   952  1073
155:   496   897  1025
156:   504   703   865
157:   533   756   925
158:   540   629   829
159:   555   572   797
160:   559   840  1009
161:   576   943  1105
162:   580   741   941
163:   585   928  1097
164:   615   728   953
165:   616   663   905
166:   620   861  1061
167:   645   812  1037
168:   660   779  1021
169:   660   989  1189
170:   696   697   985
171:   704   903  1145
172:   705   992  1217
173:   731   780  1069
174:   744   817  1105
175:   765   868  1157
176:   799   960  1249
177:   832   855  1193
178:   884   987  1325
179:   893   924  1285
```

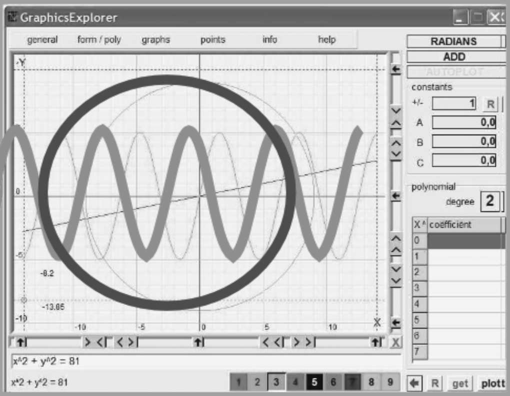
Method selection

A simple label is used as a button.
On the canvas of it's parent (`Form1`) edges are painted which change color on an enter/leave event.

A left click on the label selects the next method, a right click selects the lower method.

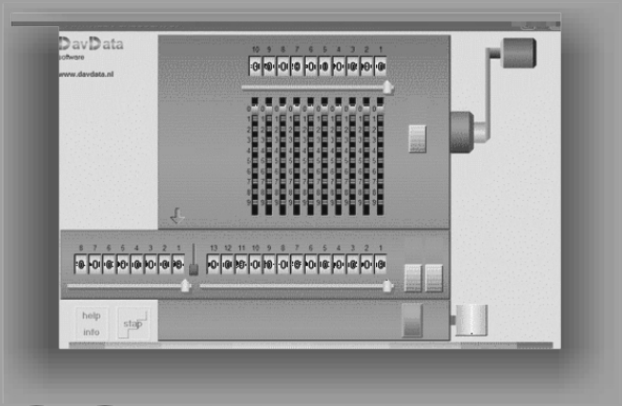
Here are pictures showing of the project at work.





x^2 + y^2 = 81

x^2 + y^2 = 81



THE BAD WAY TO PLAY CHESS:
3D PHYSICS FUN USING CASTLE GAME ENGINE (PART 1)

BY MICHALIS KAMBURELIS

Starter Expert D11

Overview of this article
❶ Introduction
❷ The Real Introduction
❸ Download and install the engine
❹ Create your first project
❺ Optionally tweak the editor preferences
❻ Learning to design 3D items in a viewport
❼ Design a 3D chessboard with chess pieces
❽ Using physics in the editor
❾ Summary

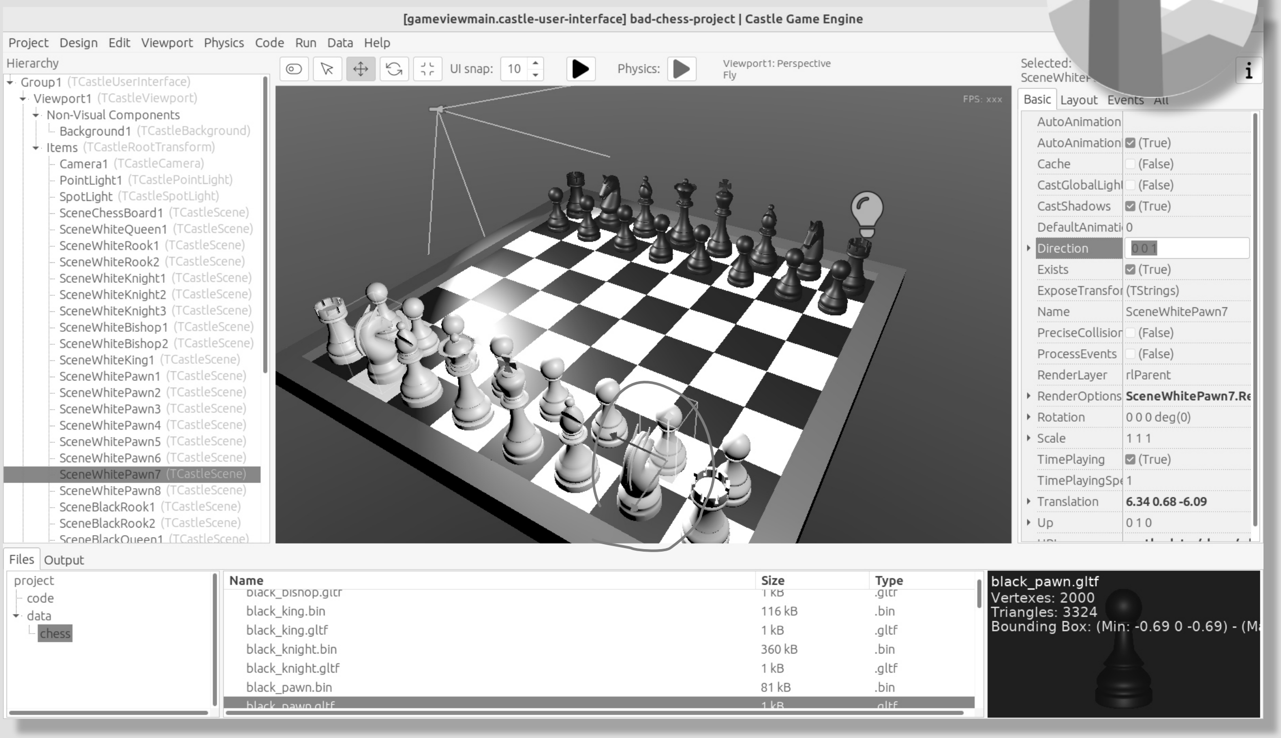
❶ INTRODUCTION

I remember my first book about chess, when I was a kid. It was a book teaching young people how to play chess. The first chapter started with a tale about children playing chess incorrectly:
they didn’t know the rules, so they put chess pieces randomly on the chessboard, and flicked them with their fingers towards the other side. The wooden chess pieces flew in the air, bashed with each other. Eventually most of the chess pieces fell off the chessboard onto the floor. The person with the last chess piece remaining on the chessboard was the winner.

That was naturally a bad way to play chess. In the second chapter of the book, an adult came, told children that they play chess wrong, and taught them the right way — how each figure moves, how the king is special, what it means to check and then mate your opponent.
The book overall was great, and it’s likely responsible for my love for chess (*the proper version of the game, with rules instead of flicking objects*) to this day.

That being said… Don’t you want to play some day this "**incorrect**" version of chess, the children’s version, where nothing else matters except just sending each chess piece flying toward the other side?

In this series of articles we will go back in time, erase our hard-earned knowledge about how to really play chess, and implement a simple 3D physics fun application where you can flick chess pieces using physics. You can treat it as a game for 2 people — just play it on a computer, and let each player use the mouse and keyboard in turn.



- Group1 (TCastleUserInterface)
  - Viewport1 (TCastleViewport)
    - Non-Visual Components
      - Background1 (TCastleBackground)
    - Items (TCastleRootTransform)
      - Camera1 (TCastleCamera)
      - PointLight1 (TCastlePointLight)
      - SpotLight (TCastleSpotLight)
      - SceneChessBoard1 (TCastleScene)
      - SceneWhiteQueen1 (TCastleScene)
      - SceneWhiteRook1 (TCastleScene)
      - SceneWhiteRook2 (TCastleScene)
      - SceneWhiteKnight1 (TCastleScene)
      - SceneWhiteKnight2 (TCastleScene)
      - SceneWhiteKnight3 (TCastleScene)
      - SceneWhiteBishop1 (TCastleScene)
      - SceneWhiteBishop2 (TCastleScene)
      - SceneWhiteKing1 (TCastleScene)
      - SceneWhitePawn1 (TCastleScene)
      - SceneWhitePawn2 (TCastleScene)
      - SceneWhitePawn3 (TCastleScene)
      - SceneWhitePawn4 (TCastleScene)
      - SceneWhitePawn5 (TCastleScene)
      - SceneWhitePawn6 (TCastleScene)
      - SceneWhitePawn7 (TCastleScene)
      - SceneWhitePawn8 (TCastleScene)
      - SceneBlackRook1 (TCastleScene)
      - SceneBlackRook2 (TCastleScene)
      - SceneBlackQueen1 (TCastleScene)

AutoAnimation	
AutoAnimation	☑ (True)
Cache	☐ (False)
CastGlobalLigh	☐ (False)
CastShadows	☑ (True)
DefaultAnimati	0
Direction	0 0 1
Exists	☑ (True)
ExposeTransfor	(TStrings)
Name	SceneWhitePawn7
PreciseCollisior	☐ (False)
ProcessEvents	☐ (False)
RenderLayer	rlParent
RenderOptions	**SceneWhitePawn7.Re**
Rotation	0 0 0 deg(0)
Scale	1 1 1
TimePlaying	☑ (True)
TimePlayingSpe	1
Translation	**6.34 0.68 -6.09**
Up	0 1 0

- project
  - code
  - data
    - chess

Name	Size	Type
black_bishop.gltf	1 kB	.gltf
black_king.bin	116 kB	.bin
black_king.gltf	1 kB	.gltf
black_knight.bin	360 kB	.bin
black_knight.gltf	1 kB	.gltf
black_pawn.bin	81 kB	.bin
black_pawn.gltf	1 kB	.gltf

black_pawn.gltf
Vertexes: 2000
Triangles: 3324
Bounding Box: (Min: -0.69 0 -0.69) - (M

❷ THE REAL INTRODUCTION

The real purpose of this article is to be an entertaining but also useful introduction to using **Castle Game Engine.**
Castle Game Engine is a cross-platform (*desktop, mobile, consoles*) 3D and 2D game engine. We will learn how to make a game for desktops (**Linux, Windows, macOS, FreeBSD**).
In the first part of the article we will show how to design a 3D chessboard and chess pieces using **Castle Game Engine** editor and how to use physics. In the next part, we will do some coding in **Pascal** to implement the game logic. In future articles we'd like to show also development for other platforms (*like Android and iOS*) and future plans (*like the web platform*).

You can use **FPC** or **Delphi** to develop the application presented here. In our engine, we are committed to perfect support for both of these **Pascal** compilers. Though note that with **Delphi**, you can right now target only **Windows** (*all the platforms are available with* **FPC**).

Castle Game Engine features a powerful visual editor to design your games, in 2D or 3D. Just like **Delphi** and **Lazarus** visual libraries, it's all based on a simple **RAD** concept: you can design a functional application easily visually but at the same time everything you do is actually using Pascal classes and properties. So all your knowledge gained from using the editor is also useful when you need to write some **Pascal** code. You will use the same classes and properties in **Pascal** that you've seen in the visual editor.

The engine is free and open-source. Use it to develop open-source or proprietary applications. You can distribute them to friends in any way, you can publish them on **Steam, Itch.io, Google Play** (**Android**), **AppStore** (**iOS**), your own website — everywhere.

❸ DOWNLOAD AND INSTALL THE ENGINE

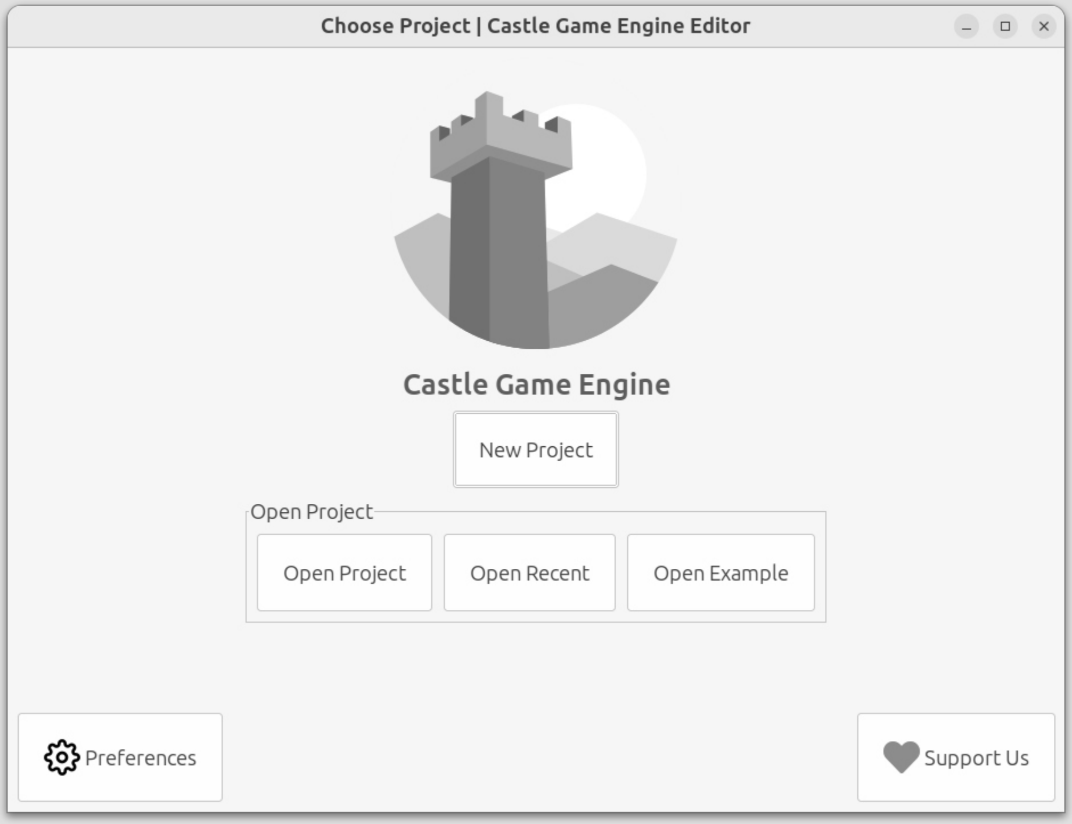
Start by downloading the engine from our website:
`https://castle-engine.io/download` .

Choose the version suitable for your operating system.

- On **Windows**, the recommended download is a simple installer. Just run it.
- On **Linux**, just unpack the downloaded zip file to any directory you like.
- Follow our website for more detailed instructions and other platforms.

Once installed, run the **Castle Game Engine** editor.

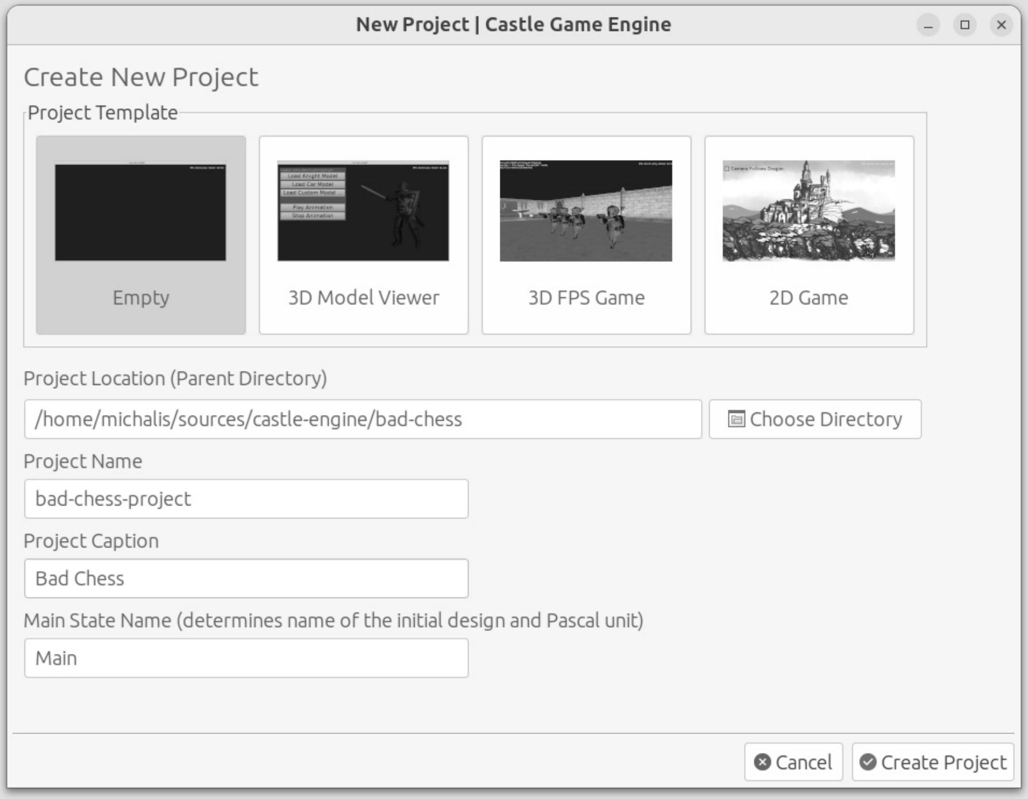
- If you used the installer on **Windows**, then the shortcut to run **Castle Game Engine** has already been created for you.
- If you unpacked the engine a zip file, then run the binary castle-editor from the subdirectory bin where you have unpacked the engine.



If you encounter any issue, consult our manual on https://castle-engine.io/install .

❹ CREATE YOUR FIRST PROJECT

Let’s create a new project. Click on the "**New Project**" button, choose the "**Empty**" project template, configure the project name and directory as you wish, and click "**Create Project**".



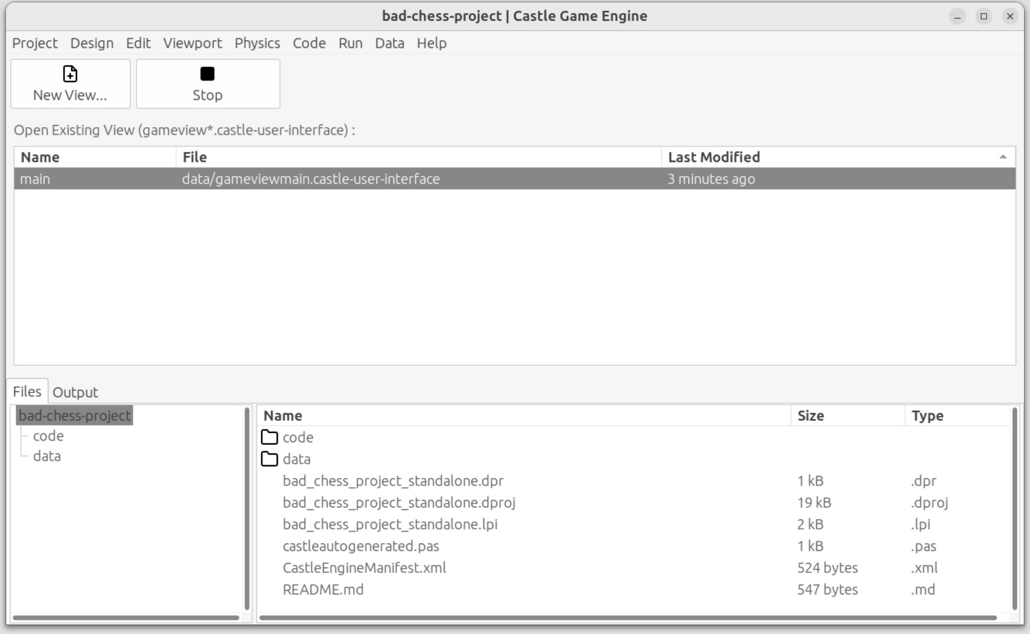
Project Location (Parent Directory): /home/michalis/sources/castle-engine/bad-chess

Project Name: bad-chess-project

Project Caption: Bad Chess

Main State Name (determines name of the initial design and Pascal unit): Main

In response, we will create a new directory with a few project files that define your project data and initial Pascal code.



Open Existing View (gameview*.castle-user-interface) :

Name	File	Last Modified
main	data/gameviewmain.castle-user-interface	3 minutes ago

bad-chess-project
- code
- data

Name	Size	Type
code		
data		
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml
README.md	547 bytes	.md

❹ CREATE YOUR FIRST PROJECT / CONTINUED

You can explore the files in your project using the bottom panel of the editor.
You can also just explore them using your regular file manager — there's nothing special about this directory, these are normal files and directories.

The most important files and directories are:

- `code` is a subdirectory where we advise to put all **Pascal** source code (units) of your application. Initially it contains just 2 units, `GameInitialize` and `GameViewMain`.

- `data` is a subdirectory where you should put all the data that has to be loaded at run-time by your application. All the 3D and 2D models, textures, designs have to be placed here if you want to use them in your game. Initially it contains the design called `gameviewmain.castle-user-interface` (*and, less important,* `CastleSettings.xml` *and* `README.txt` *files*).

The general idea is that the initial application (*created from the "Empty" template*) contains just a single view called `Main`. A view is a **Castle Game Engine** concept that represents something that can be displayed in a **Castle Game Engine** application.
You use it typically quite like a form in **Delphi or Lazarus.** It is a basic way to organize your application.

- Every view can be visually designed. Just double-click on it, in the "**Open Existing View**" panel or in the "**Files**" panel (*when you're exploring the* `data` *subdirectory*).

  This allows to visually design the contents of the `gameviewmain.castle-user-interface` file. The file has an extension `.castle-user-interface` because a view is a special case of user interface in **Castle Game Engine**.

  In larger applications, you can have multiple views. Also, in larger applications, you can visually design some user interface elements that are not views, but are just reusable pieces of a user interface. All these files have the extension `.castle-user-interface` and can be visually designed using the editor.
  The views have, by convention, a name like `gameview*.castle-user-interface.`

- Every view has also an accompanying **Pascal** unit. The unit is named like the view, but without the `.castle-user-interface` extension. So in our case, the unit is called `gameviewmain.pas.` The unit contains the **Pascal** code that should be executed when the view is displayed. It defines a class that has virtual methods to react to various useful events (*like view being started, or user pressing a key or a mouse button*).
  You will often add more methods to it, to implement your application logic.

See **`https://castle-engine.io/view_events`** and **`https://castle-engine.io/views`** to learn more about the views in our engine.

To be clear about the terminology used throughout our engine:

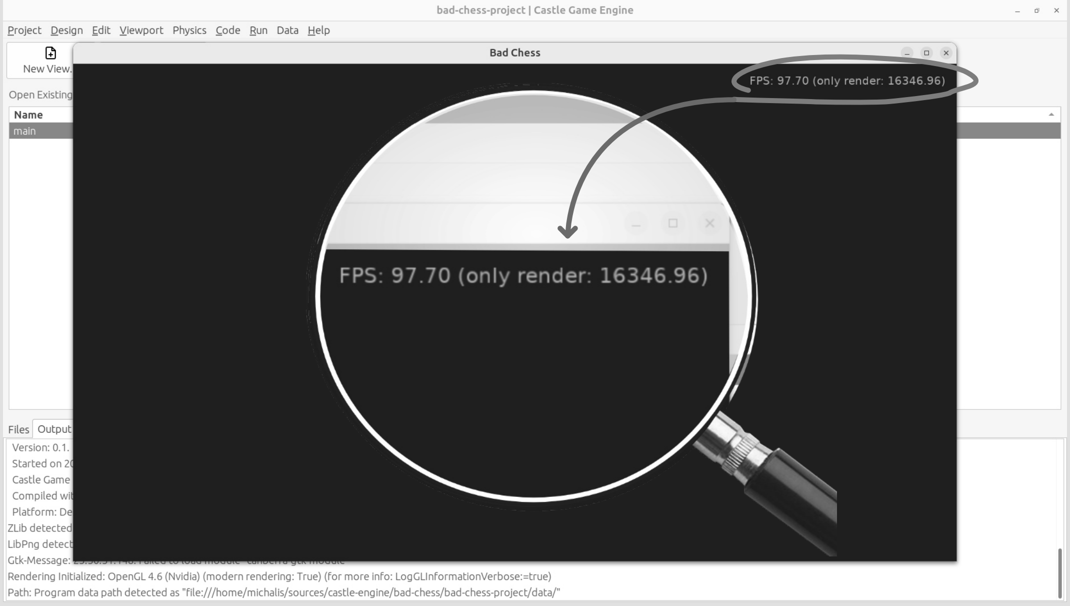
- A design is a name for a file you can visually design using our editor.
  A design can be a file with extension:
  - .castle-user-interface (*user interface, can be loaded to a class descending from* `TCastleUserInterface`)
  - .castle-transform (*3D or 2D transformation, can be loaded to a class descending from* `TCastleTransform`)
  - .castle-component (*any other component; can be loaded to a class descending from* `TComponent`)

❹ CREATE YOUR FIRST PROJECT / CONTINUED

- A user interface design is a specific case of a design file. It is a file with `.castle-user-interface` extension.
- A view is a specific case of a user interface design. By convention it is called like `gameview*.castle-user-interface`.

You're probably itching to start actually doing something after this lengthy introduction. Let's get to it.

As a first thing, make sure that everything works. Use the big **"Compile And Run"** button (**key shortcut F9**) and watch as the project is compiled and run. The result will be boring — dark window with **FPS** (*frames per second*) counter in the top-right corner. **FPS** are a standard way to measure your application performance.



```
FPS: 97.70 (only render: 16346.96)
```

```
Version: 0.1.
Started on 20
Castle Game
Compiled wit
Platform: De
ZLib detected
LibPng detect
Gtk-Message:
Rendering Initialized: OpenGL 4.6 (Nvidia) (modern rendering: True) (for more info: LogGLInformationVerbose:=true)
Path: Program data path detected as "file:///home/michalis/sources/castle-engine/bad-chess/bad-chess-project/data/"
```

❺ OPTIONALLY TWEAK THE EDITOR PREFERENCES

Once things work, you may want to tweak them by going to editor "Preferences". In particular:

- The editor by default uses a bundled version of latest stable **FPC** (**Free Pascal Compiler**). If you'd rather use your own **FPC** installation or **Delphi**, configure it in the preferences.
- To edit the Pascal files, the editor by default tries to auto-detect various Pascal-capable IDEs and editors, like **Lazarus, Delphi, Visual Studio Code**. If you prefer to configure a specific editor, choose it in the preferences.

More details about the editor configuration can be found in our manual on `https://castle-engine.io/install` .

The editor can use any **Pascal** compiler and any text editor. We deliberately don't put any special requirements on what you can use. Though we make sure to support the popular choices perfectly. In particular, we have a dedicated support for using **Visual Studio Code with Pascal** (and **Castle Game Engine** in particular), see `https://castle-engine.io/vscode` .

❻ LEARNING TO DESIGN 3D ITEMS IN A VIEWPORT

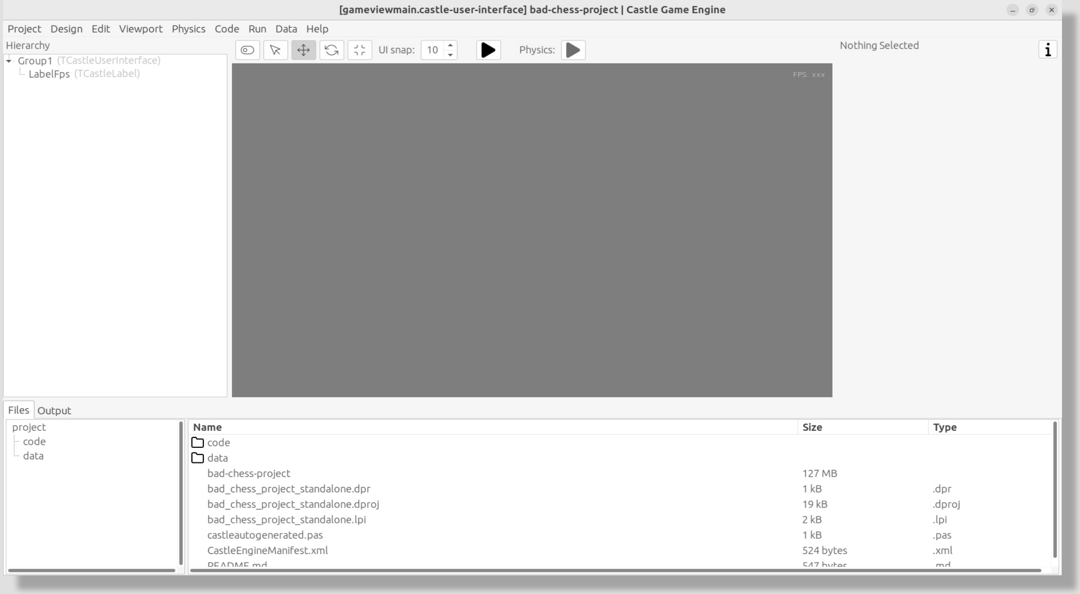
If you haven't already, open the main view in the editor.

You can double-click on it in the **"Open Existing View"** panel or in the "Files" panel (*when you're exploring the data subdirectory*).

The initial view is mostly empty.

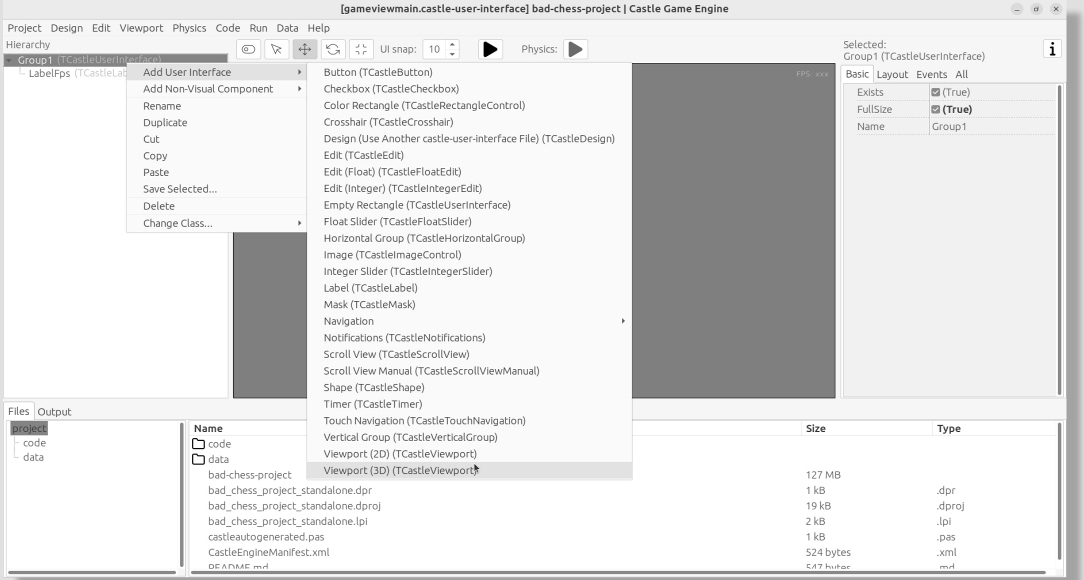
It has a root component Group1, which is an instance of `TCastleUserInterface`. This component will contain everything else we design.

And it has a label `LabelFps` (*an instance of* `TCastleLabel` *class*). At run-time, this label will display the **FPS** counter.



Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml

Let's add more content to it. First of all, to display anything in 3D, you need a viewport. A viewport is a way to display 3D or 2D content. It is an instance of `TCastleViewport` class. Add it to the design by **right-clicking** on the `Group1` component and choosing **"Add User Interface → Viewport (3D)"** from the menu that appears.

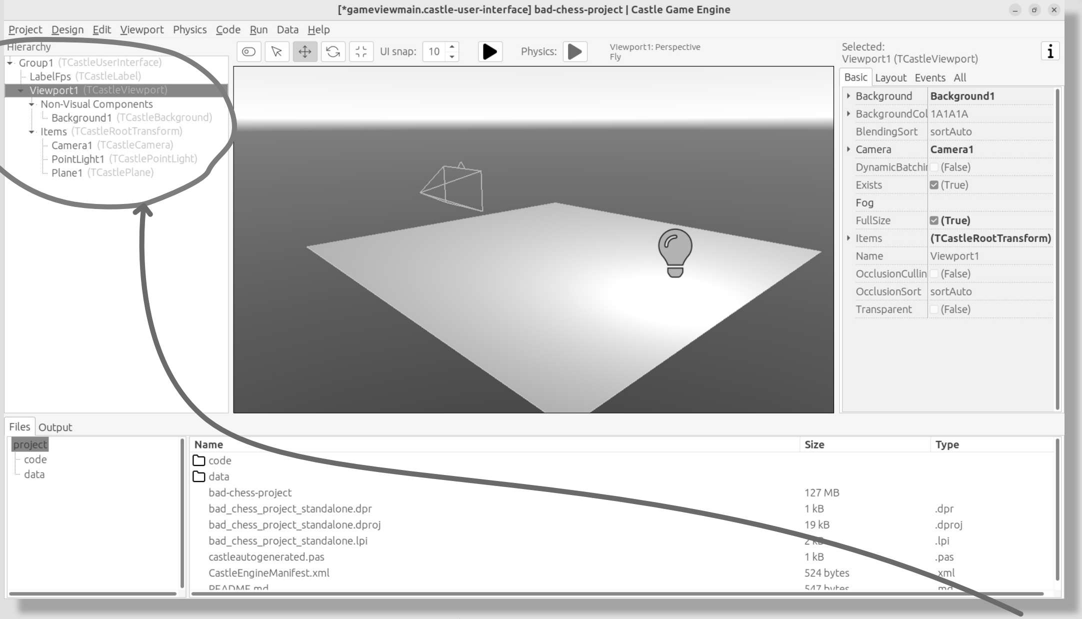


Exists	☑ (True)
FullSize	☑ **(True)**
Name	Group1

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml

❻ LEARNING TO DESIGN 3D ITEMS IN A VIEWPORT / CONTINUATION

The result should look like this:



Hierarchy:
- Group1 (TCastleUserInterface)
  - LabelFps (TCastleLabel)
  - Viewport1 (TCastleViewport)
    - Non-Visual Components
      - Background1 (TCastleBackground)
    - Items (TCastleRootTransform)
      - Camera1 (TCastleCamera)
      - PointLight1 (TCastlePointLight)
      - Plane1 (TCastlePlane)

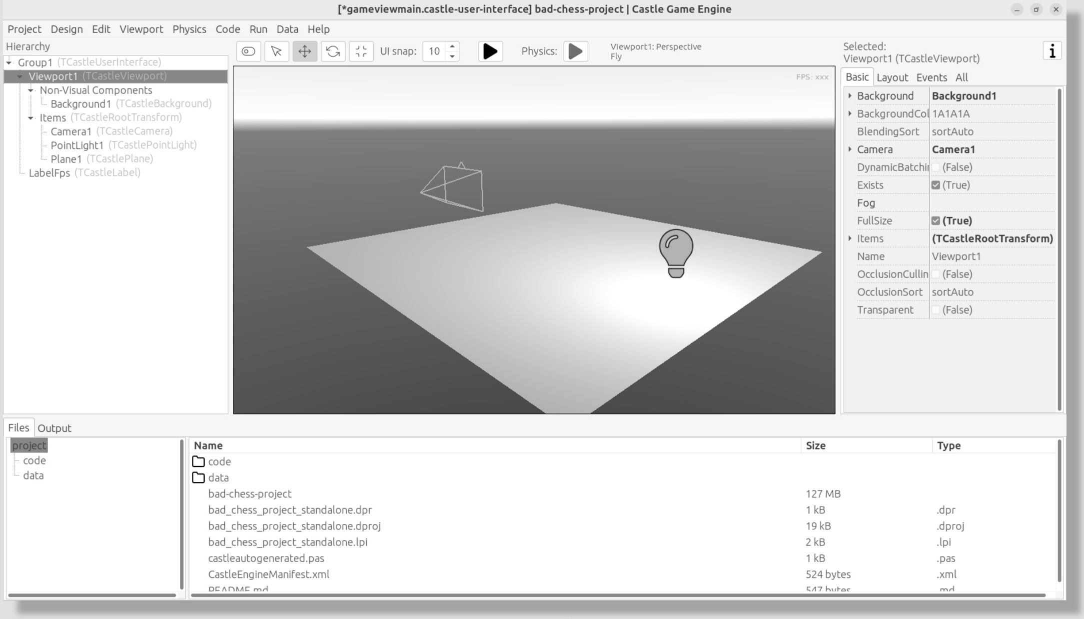
Selected:
Viewport1 (TCastleViewport)

Background	**Background1**
BackgroundCol	1A1A1A
BlendingSort	sortAuto
Camera	**Camera1**
DynamicBatchir	☐ (False)
Exists	☑ (True)
Fog	
FullSize	☑ **(True)**
Items	**(TCastleRootTransform)**
Name	Viewport1
OcclusionCullin	☐ (False)
OcclusionSort	sortAuto
Transparent	☐ (False)

project
- code
- data

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml

Following this, drag the new Viewport1 component above the `LabelFps` in the **Hierarchy** panel (*on the left*). This way the FPS counter will be displayed in front of the viewport.



Hierarchy:
- Group1 (TCastleUserInterface)
  - Viewport1 (TCastleViewport)
    - Non-Visual Components
      - Background1 (TCastleBackground)
    - Items (TCastleRootTransform)
      - Camera1 (TCastleCamera)
      - PointLight1 (TCastlePointLight)
      - Plane1 (TCastlePlane)
  - LabelFps (TCastleLabel)

Selected:
Viewport1 (TCastleViewport)

Background	**Background1**
BackgroundCol	1A1A1A
BlendingSort	sortAuto
Camera	**Camera1**
DynamicBatchir	☐ (False)
Exists	☑ (True)
Fog	
FullSize	☑ **(True)**
Items	**(TCastleRootTransform)**
Name	Viewport1
OcclusionCullin	☐ (False)
OcclusionSort	sortAuto
Transparent	☐ (False)

project
- code
- data

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml

Now play around in the 3D view. There are 3 objects in 3D world:

Camera, called just Camera1, determines what the user will actually see once the game is run.

Light source makes things lit (bright). The initial light source is called PointLight1 and it is an instance of `TCastlePointLight`, which is a simple light that shines in all directions from a given 3D position.

Rectangle representing a ground called a **Plane1**.
Mathematically speaking, it's not a plane, it's a rectangle — however calling this a "**plane**" is a convention used by a lot of 3D software.

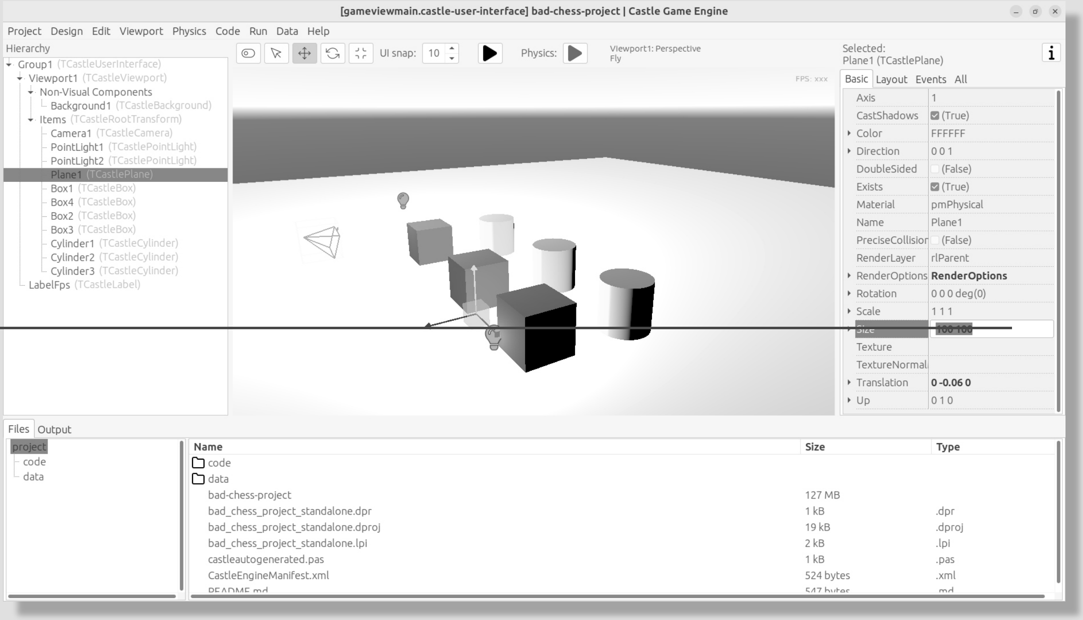
Click and hold the right mouse button over the viewport to look around.

Use the **AWSD** keys to move. Use the mouse scroll (*while holding the right mouse button pressed*) to increase or decrease the movement speed.

Play around with moving the items. Drag the 3D axis to move any object.

Play around with adding new 3D items. **Right-click** on Items component inside the **Viewport1** and from the context menu add primitives like **"Box", "Sphere", "Cylinder".**

Move them around, delete them (with **Delete** key), duplicate (*with* **Ctrl+D** *key*). Change some properties. On the right side, you can see an object inspector, familiar to any **Lazarus** and **Delphi** user. Adjust the properties, for example change the Size of the **Plane1** to be much bigger. Click on "**...**" (*3 dots, called* **Ellipsis**) button at the "**Color**" property of any primitive (*like a plane, a box, a sphere...*) to change the color.



Axis	1
CastShadows	☑ (True)
Color	FFFFFF
Direction	0 0 1
DoubleSided	☐ (False)
Exists	☑ (True)
Material	pmPhysical
Name	Plane1
PreciseCollisior	☐ (False)
RenderLayer	rlParent
RenderOptions	**RenderOptions**
Rotation	0 0 0 deg(0)
Scale	1 1 1
Size	100 100
Texture	
TextureNormal	
Translation	**0 -0.06 0**
Up	0 1 0

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	524 bytes	.xml

If you get stuck, consult our manual, in particular `https://castle-engine.io/viewport_and_scenes` and `https://castle-engine.io/viewport_3d` are helpful to learn basic 3D manipulation.

7. DESIGN A 3D CHESSBOARD WITH CHESS PIECES

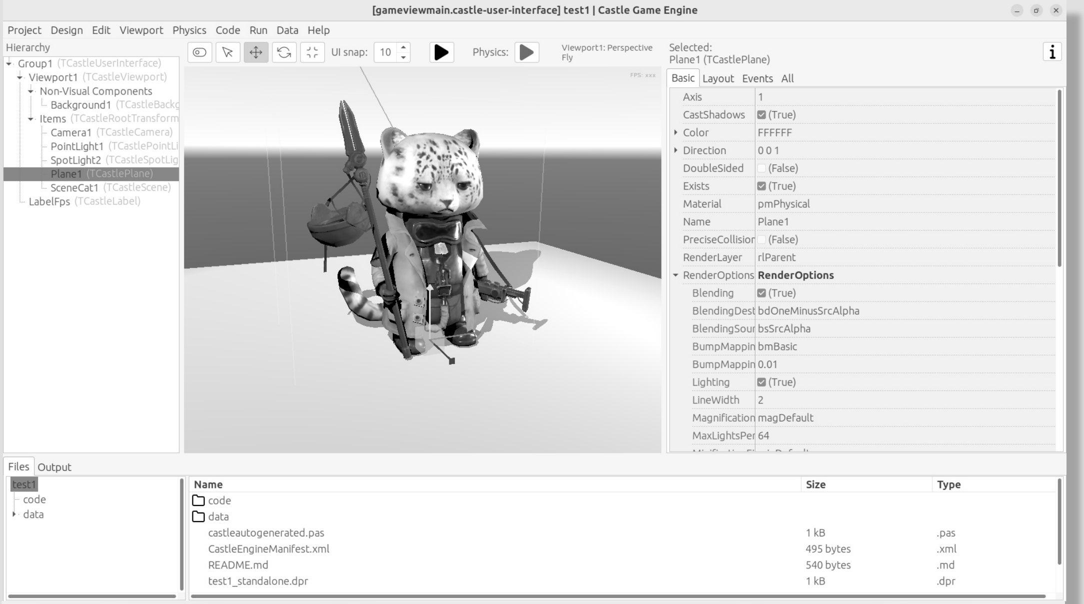
Above we learned to design a 3D world composed from simple primitives, like boxes and spheres.
But this isn't a way to create realistic 3D graphics. In most 3D graphic applications, the content is created using a specialized 3D authoring tool, like Blender. 3D artist creates a mesh (*a set of vertexes, connected to form edges and polygons*), assigns materials and textures, and exports the resulting object to a file that can be read by a game engine — like a `glTF` **(*1)** file.
Castle Game Engine has great support for `glTF`. See `https://castle-engine.io/gltf`

On **Castle Game Engine** side, our most important component to display a 3D model is `TCastleScene`. It's a big component, playing central role in our engine (*in one way or another, it is actually responsible for all of 3D and 2D rendering in our viewport*). Using it is simple: you create an instance of `TCastleScene` and set its URL property to point to the model you want to display (*like a* **glTF** *file*). The `TCastleScene` class descends from the `TCastleTransform` class, and as such you can move, rotate and scale the `TCastleScene` instances. Alternatively, you can also drag-and-drop the `glTF` file from the "**Files**" panel to the viewport, editor will then automatically create a `TCastleScene` instance that loads the given model.

We support a number of 3D and 2D model formats, not only glTF. They are listed on `https://castle-engine.io/creating_data_model_formats.php` .
If you are capable of creating your own 3D models, for example in Blender, you can now make a detour: design a 3D model in Blender and export it to **glTF** using our instructions on `https://castle-engine.io/blender` .
Or you can use some ready-made stuff:

- There's a number of high-quality 3D content on the Internet, available also for free and on open-source-compatible licenses. We collect useful links on `https://castle-engine.io/assets.php` .
- Our engine also features an integration with **Sketchfab**, to allow you to search and download from a vast repository of free 3D models without leaving our editor. See the `https://castle-engine.io/sketchfab documentation.`

Here's a sample — battle-hardened cat model, from **Sketchfab**, right inside our editor:



Selected:
Plane1 (TCastlePlane)

Axis	1
CastShadows	☑ (True)
Color	FFFFFF
Direction	0 0 1
DoubleSided	☐ (False)
Exists	☑ (True)
Material	pmPhysical
Name	Plane1
PreciseCollisior	☐ (False)
RenderLayer	rlParent
RenderOptions	**RenderOptions**
Blending	☑ (True)
BlendingDest	bdOneMinusSrcAlpha
BlendingSour	bsSrcAlpha
BumpMappin	bmBasic
BumpMappin	0.01
Lighting	☑ (True)
LineWidth	2
Magnification	magDefault
MaxLightsPer	64

Name	Size	Type
code		
data		
castleautogenerated.pas	1 kB	.pas
CastleEngineManifest.xml	495 bytes	.xml
README.md	540 bytes	.md
test1_standalone.dpr	1 kB	.dpr

Credits: The "Cat" 3D model was done by Muru (`https://sketchfab.com/muru`) and is available on Sketchfab (`https://sketchfab.com/3d-models/cat-16c3444c8d1440fc97fdf10f60ec58b0`) on CC-BY-4.0 license.

- Finally, we have a ready set of 3D models for the chessboard and all chess pieces, that you can use for this demo.

To use the last option, download the 3D models from `https://github.com/castle-engine/bad-chess/releases/download/chess-models/chess-models.zip` .
They were made based on open-source Blender model published on `https://blendswap.com/blend/29244` by **Phuong2647**.

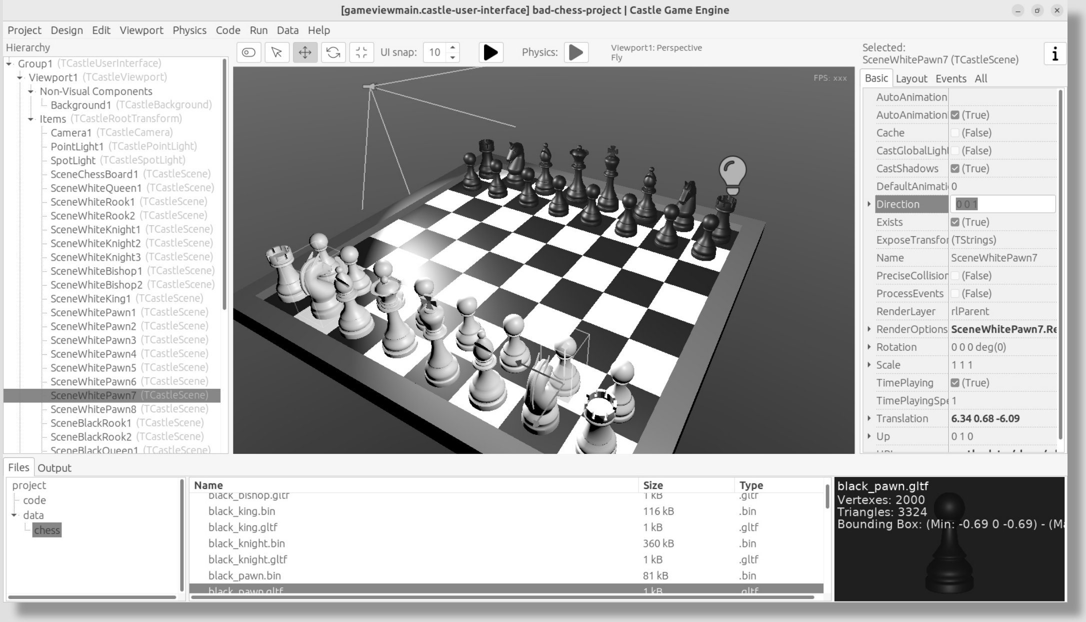
Unpack the resulting archive anywhere under the data subdirectory of your project.

Then simply drag-and-drop the `*.gltf` files onto the viewport. Move and duplicate them as needed, to arrange them into a starting chess position.

NOTE
For our silly physics game, it actually completely doesn’t matter how you will arrange them. You also don’t need to position and rotate them perfectly. *Have fun :)*

WIKIPEDIA
`glTF` (*1) is a standard file format for three-dimensional scenes and models. A `glTF` file uses one of two possible file extensions: `.gltf` (JSON/ASCII) or `.glb` (*binary*). Both `.gltf` and `.glb` files may reference external binary and texture resources. Alternatively, both formats may be self-contained by directly embedding binary data buffers (*as base64-encoded strings in .gltf files or as raw byte arrays in .glb files*).
An open standard developed and maintained by the **Khronos Group**, it supports 3D model geometry, appearance, scene graph hierarchy, and animation. It is intended to be a streamlined, inter-operable format for the delivery of 3D assets, while minimizing file size and runtime processing by apps. As such, its creators have described it as the "**JPEG of 3D**."

This is an example result:



Group1 (TCastleUserInterface)
  Viewport1 (TCastleViewport)
    Non-Visual Components
      Background1 (TCastleBackground)
    Items (TCastleRootTransform)
      Camera1 (TCastleCamera)
      PointLight1 (TCastlePointLight)
      SpotLight (TCastleSpotLight)
      SceneChessBoard1 (TCastleScene)
      SceneWhiteQueen1 (TCastleScene)
      SceneWhiteRook1 (TCastleScene)
      SceneWhiteRook2 (TCastleScene)
      SceneWhiteKnight1 (TCastleScene)
      SceneWhiteKnight2 (TCastleScene)
      SceneWhiteKnight3 (TCastleScene)
      SceneWhiteBishop1 (TCastleScene)
      SceneWhiteBishop2 (TCastleScene)
      SceneWhiteKing1 (TCastleScene)
      SceneWhitePawn1 (TCastleScene)
      SceneWhitePawn2 (TCastleScene)
      SceneWhitePawn3 (TCastleScene)
      SceneWhitePawn4 (TCastleScene)
      SceneWhitePawn5 (TCastleScene)
      SceneWhitePawn6 (TCastleScene)
      SceneWhitePawn7 (TCastleScene)
      SceneWhitePawn8 (TCastleScene)
      SceneBlackRook1 (TCastleScene)
      SceneBlackRook2 (TCastleScene)
      SceneBlackQueen1 (TCastleScene)

Selected:
SceneWhitePawn7 (TCastleScene)

AutoAnimation	
AutoAnimation	☑ (True)
Cache	☐ (False)
CastGlobalLigh	☐ (False)
CastShadows	☑ (True)
DefaultAnimati	0
Direction	0 0 1
Exists	☑ (True)
ExposeTransfor	(TStrings)
Name	SceneWhitePawn7
PreciseCollisior	☐ (False)
ProcessEvents	☐ (False)
RenderLayer	rlParent
RenderOptions	**SceneWhitePawn7.Re**
Rotation	0 0 0 deg(0)
Scale	1 1 1
TimePlaying	☑ (True)
TimePlayingSpe	1
Translation	**6.34 0.68 -6.09**
Up	0 1 0

project
  code
  data
    chess

Name	Size	Type
black_bishop.gltf	1 kB	.gltf
black_king.bin	116 kB	.bin
black_king.gltf	1 kB	.gltf
black_knight.bin	360 kB	.bin
black_knight.gltf	1 kB	.gltf
black_pawn.bin	81 kB	.bin
black_pawn.gltf	1 kB	.gltf

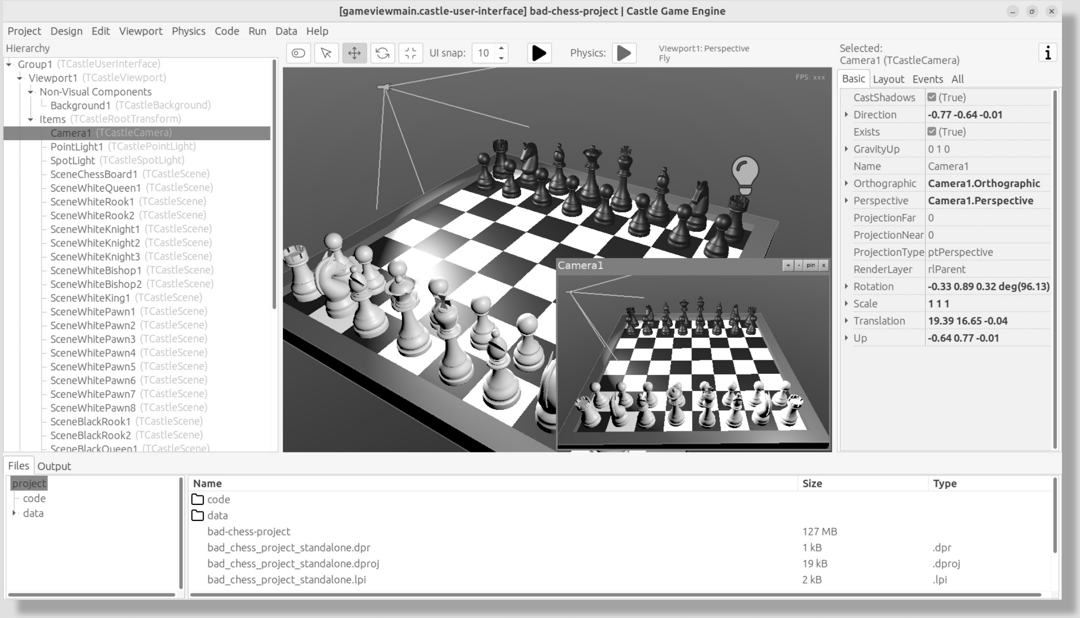
black_pawn.gltf
Vertexes: 2000
Triangles: 3324
Bounding Box: (Min: -0.69 0 -0.69) - (M

Once you've designed the chessboard and put chess pieces on it, also make sure to adjust the lights to make everything nicely bright (*but not too bright*).

Finally, adjust the camera so that user sees a nice view of the board when the application starts. When you select a camera component (*like* **Camera1**, *if you haven't renamed the default camera*), the editor shows a small window with camera preview. You can click "**Pin**" in this window to keep observing the world from this camera. There are basically 2 ways to manipulate the camera:

❶ Move and rotate the camera just like any other 3D object. Look at the camera preview to judge whether the camera view looks good.

❷ Or, alternatively, navigate in the editor and then use the menu item "**Viewport → Align Camera To View**" (**key shortcut Ctrl + Numpad 0**) to make the camera view match the current view in the editor.



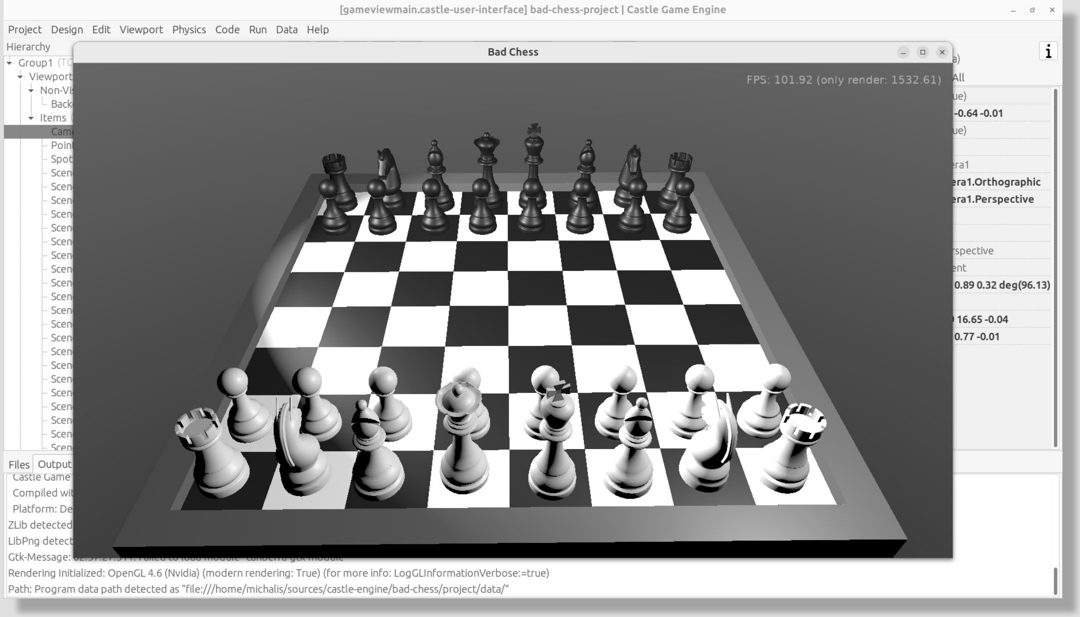
Selected:
Camera1 (TCastleCamera)

Basic | Layout | Events | All

CastShadows	☑ (True)
Direction	**-0.77 -0.64 -0.01**
Exists	☑ (True)
GravityUp	0 1 0
Name	Camera1
Orthographic	**Camera1.Orthographic**
Perspective	**Camera1.Perspective**
ProjectionFar	0
ProjectionNear	0
ProjectionType	ptPerspective
RenderLayer	rlParent
Rotation	**-0.33 0.89 0.32 deg(96.13)**
Scale	**1 1 1**
Translation	**19.39 16.65 -0.04**
Up	**-0.64 0.77 -0.01**

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi

Once you have a nice view, make sure it all works: compile and run the application again.



```
Rendering Initialized: OpenGL 4.6 (Nvidia) (modern rendering: True) (for more info: LogGLInformationVerbose:=true)
Path: Program data path detected as "file:///home/michalis/sources/castle-engine/bad-chess/project/data/"
```

❽ USING PHYSICS IN THE EDITOR

Now that the proper chessboard with chess pieces is designed, let’s use physics to make things crazier.

Castle Game Engine has a support for rigid body physics. This means that:

- Objects can be affected by forces.
  The force that works automatically is gravity, pulling objects down (*in the direction of the negative Y axis, by default*).

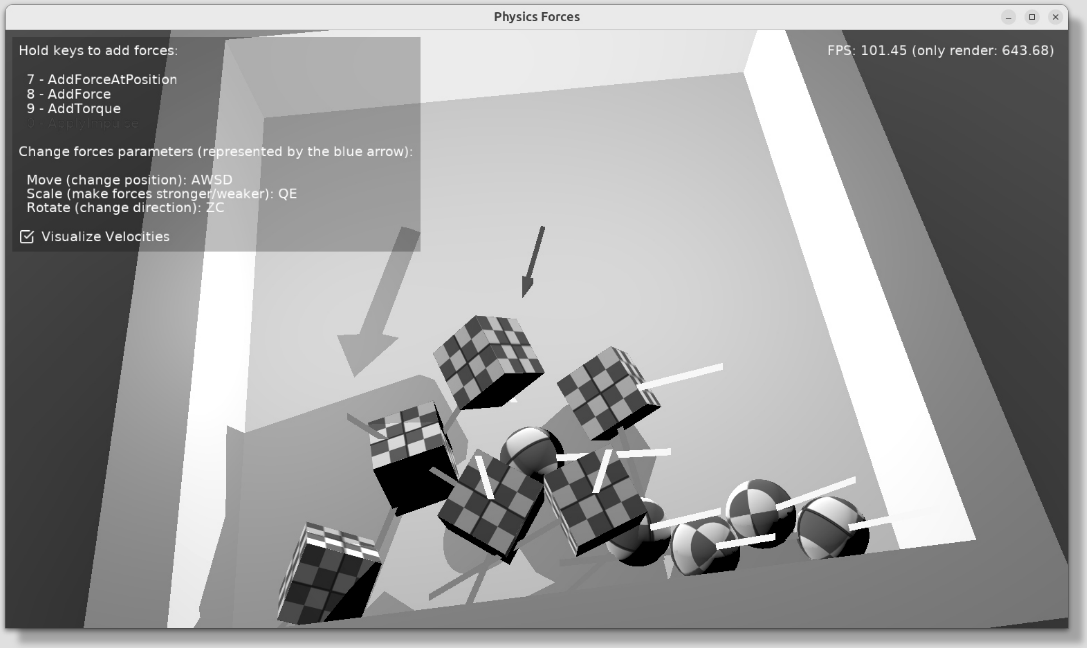
  You can also define additional forces from code, to e.g. push things along an arbitrary direction. Your own forces can realize a range of real-life effects, like wind, explosions, spinning tornadoes, etc.

- Collisions between objects are automatically detected and resolved.
  That is, by default the objects will bounce off each other.

  It is also possible to detect collisions in code and react to them in any way (*e.g. an enemy may explode when it collides with a rocket*).

- You can also connect certain objects using joints.

We will not explore all these features in our article, but we will show you how to enjoy the basics. To learn more about the possibilities, read our manual `https://castle-engine.io/physics` and play with demo's in the **`examples/physics/`** subdirectory of the engine. Here’s a screenshot from one of the demos, showing explicit application of physics forces:



```
Hold keys to add forces:

  7 - AddForceAtPosition
  8 - AddForce
  9 - AddTorque
  0 - ApplyImpulse

Change forces parameters (represented by the blue arrow):

  Move (change position): AWSD
  Scale (make forces stronger/weaker): QE
  Rotate (change direction): ZC

☑ Visualize Velocities
```

```
FPS: 101.45 (only render: 643.68)
```

Castle Game Engine physics internally uses Kraft, a physics engine developed in **Pascal** by **Benjamin 'BeRo' Rosseaux.**

Any component descending `TCastleTransform`, including primitives (like `TCastleBox`) or scenes loaded from models (`TCastleScene`) or a group of other objects (`TCastleTransform` *with children*) can be a rigid body for the physics engine that participates in the collision detection and resulting movement. The object needs to have two behaviors:

❶ `TCastleRigidBody` behavior makes the component a rigid body.
It defines common physics properties, like whether the object is affected by gravity and the initial movement speed.

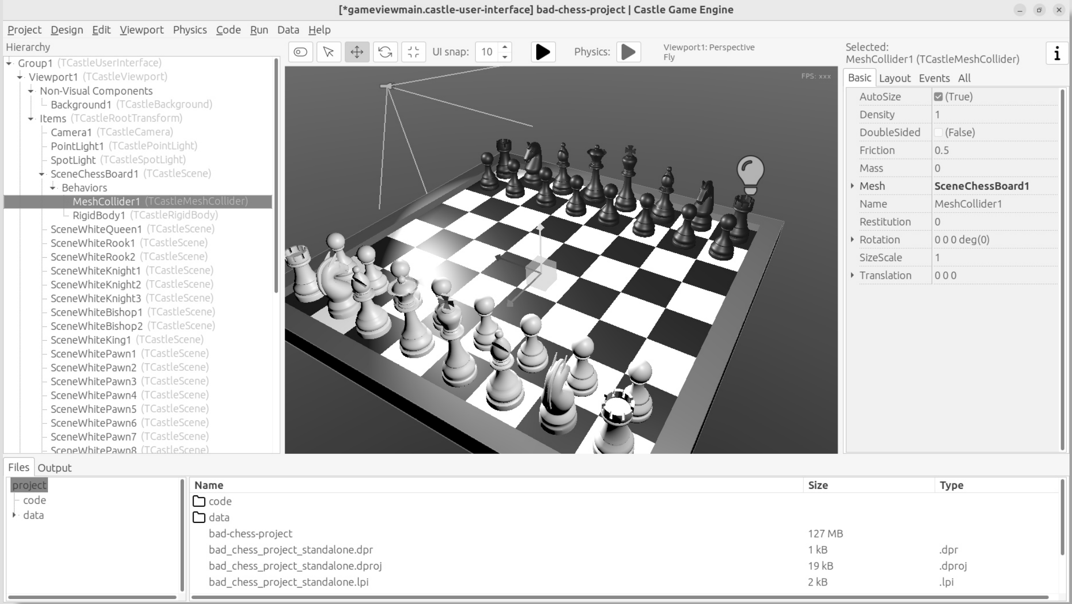
❷ A collider, which stands for any component descending from the abstract class `TCastleCollider`. Many collider shapes are possible, like `TCastleSphereCollider`, `TCastleBoxCollider` and `TCastleMeshCollider`.

Using the `TCastleMeshCollider` results in most precise collisions, but the colliding object must be static which means that other objects will bounce off this object, but the object with `TCastleMeshCollider` will not move itself.

The term behavior we used above is a special mechanism in Castle Game Engine to attach additional functionality to a TCastleTransform. Behaviors are a great way to define various functionality that enhances given game object. There are various built-in behaviors and you can also define your own.
See **https://castle-engine.io/behaviors** for more information.

After this overview, you're ready to actually use physics in our chess game.

Right-click on the component representing the chessboard. From the context menu choose "**Add Behavior (Extends Parent Transform) → Physics → Collider → Mesh**".
In response, you will notice that 2 components have appeared in the component tree: **MeshCollider1** and **RigidBody1**.
That's a convenience feature of the editor: adding a collider also adds a rigid body component.



```
Group1 (TCastleUserInterface)
  Viewport1 (TCastleViewport)
    Non-Visual Components
      Background1 (TCastleBackground)
    Items (TCastleRootTransform)
      Camera1 (TCastleCamera)
      PointLight1 (TCastlePointLight)
      SpotLight (TCastleSpotLight)
      SceneChessBoard1 (TCastleScene)
        Behaviors
          MeshCollider1 (TCastleMeshCollider)
          RigidBody1 (TCastleRigidBody)
      SceneWhiteQueen1 (TCastleScene)
      SceneWhiteRook1 (TCastleScene)
      SceneWhiteRook2 (TCastleScene)
      SceneWhiteKnight1 (TCastleScene)
      SceneWhiteKnight2 (TCastleScene)
      SceneWhiteKnight3 (TCastleScene)
      SceneWhiteBishop1 (TCastleScene)
      SceneWhiteBishop2 (TCastleScene)
      SceneWhiteKing1 (TCastleScene)
      SceneWhitePawn1 (TCastleScene)
      SceneWhitePawn2 (TCastleScene)
      SceneWhitePawn3 (TCastleScene)
      SceneWhitePawn4 (TCastleScene)
      SceneWhitePawn5 (TCastleScene)
      SceneWhitePawn6 (TCastleScene)
      SceneWhitePawn7 (TCastleScene)
      SceneWhitePawn8 (TCastleScene)
```

Selected:
MeshCollider1 (TCastleMeshCollider)

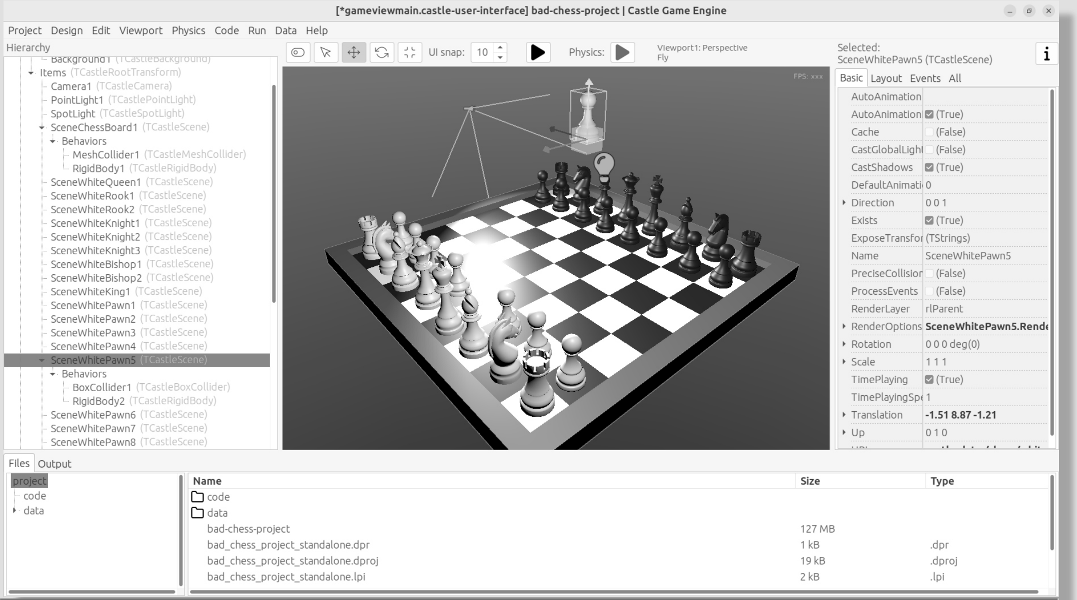
AutoSize	☑ (True)
Density	1
DoubleSided	☐ (False)
Friction	0.5
Mass	0
Mesh	**SceneChessBoard1**
Name	MeshCollider1
Restitution	0
Rotation	0 0 0 deg(0)
SizeScale	1
Translation	0 0 0

project
- code
- data

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi

Next choose any chess piece. **Right-click** on it and from the context menu choose **"Add Behavior (Extends Parent Transform) → Physics → Collider → Box"**. Note that we use a simpler collider for the chess piece, which is also dynamic. This will allow the chess piece to actually fall down on the board.

Finally move the chess piece to a more dramatic position, above the board, so that it will fall down when the physics will start.



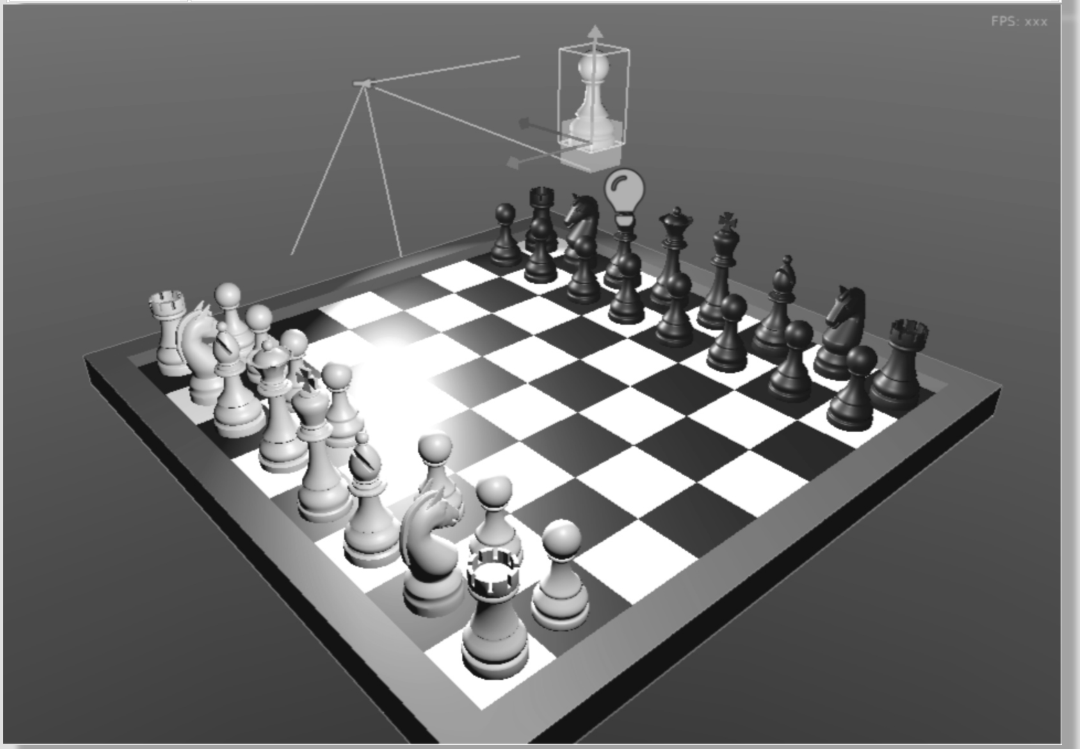
Background1 (TCastleBackground)
Items (TCastleRootTransform)
  Camera1 (TCastleCamera)
  PointLight1 (TCastlePointLight)
  SpotLight (TCastleSpotLight)
  SceneChessBoard1 (TCastleScene)
    Behaviors
      MeshCollider1 (TCastleMeshCollider)
      RigidBody1 (TCastleRigidBody)
  SceneWhiteQueen1 (TCastleScene)
  SceneWhiteRook1 (TCastleScene)
  SceneWhiteRook2 (TCastleScene)
  SceneWhiteKnight1 (TCastleScene)
  SceneWhiteKnight2 (TCastleScene)
  SceneWhiteKnight3 (TCastleScene)
  SceneWhiteBishop1 (TCastleScene)
  SceneWhiteBishop2 (TCastleScene)
  SceneWhiteKing1 (TCastleScene)
  SceneWhitePawn1 (TCastleScene)
  SceneWhitePawn2 (TCastleScene)
  SceneWhitePawn3 (TCastleScene)
  SceneWhitePawn4 (TCastleScene)
  SceneWhitePawn5 (TCastleScene)
    Behaviors
      BoxCollider1 (TCastleBoxCollider)
      RigidBody2 (TCastleRigidBody)
  SceneWhitePawn6 (TCastleScene)
  SceneWhitePawn7 (TCastleScene)
  SceneWhitePawn8 (TCastleScene)

Selected:
SceneWhitePawn5 (TCastleScene)

AutoAnimation	
AutoAnimation	☑ (True)
Cache	☐ (False)
CastGlobalLight	☐ (False)
CastShadows	☑ (True)
DefaultAnimati	0
Direction	0 0 1
Exists	☑ (True)
ExposeTransfor	(TStrings)
Name	SceneWhitePawn5
PreciseCollision	☐ (False)
ProcessEvents	☐ (False)
RenderLayer	rlParent
RenderOptions	**SceneWhitePawn5.Rende**
Rotation	0 0 0 deg(0)
Scale	1 1 1
TimePlaying	☑ (True)
TimePlayingSpe	1
Translation	**-1.51 8.87 -1.21**
Up	0 1 0

project
  code
  data

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi



We are ready to run physics. One way would be to just run the application, using the "**Compile And Run**" as you've done before. But there's a quicker way to experiment with physics: run **physics simulation** by using the green play icon at the header of the editor (**or menu item "Physics → Play Simulation", key shortcut Ctrl+P**).

Do this and watch in awe as the pawn falls on the board.

Remember to finish the physics simulation when you're done (*press the green stop button, or again menu item* **"Physics → Play Simulation", key shortcut Ctrl+P**).
Editing the design during the physics simulation is allowed (*and it's a great way to experiment with various physics settings*) but the changes are not saved when physics simulation is running. That's because physics typically moves the objects, and you don't want to save this position resulting from physics interactions. So be sure to stop the physics simulation **before doing any persistent changes** to the design.

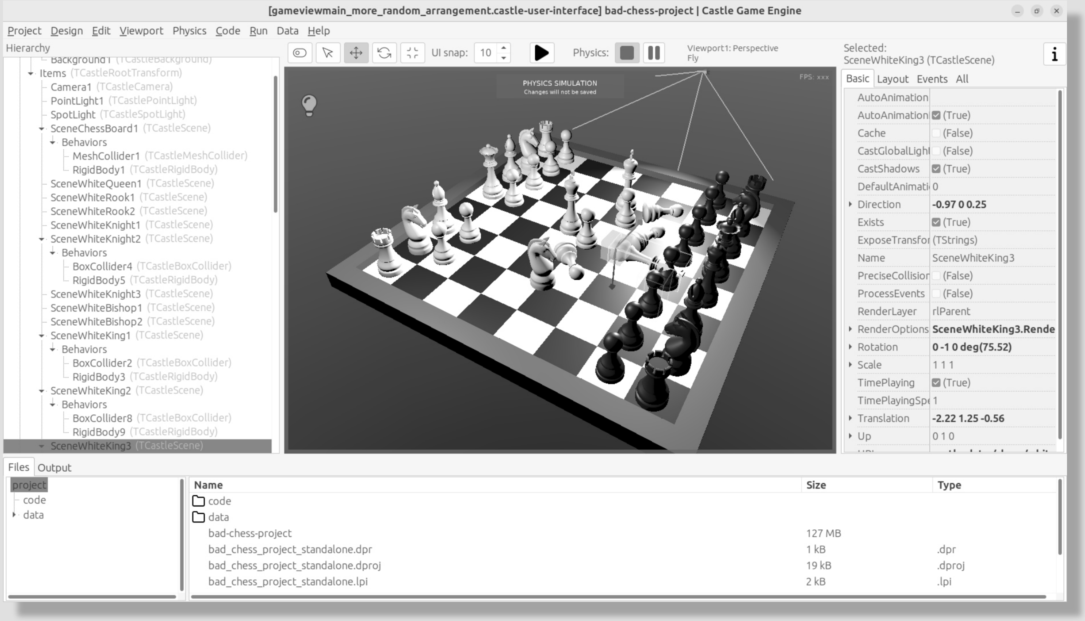
To get more spectacular results:

- Add **physics colliders** to more chess pieces.
- Move the chess pieces to more interesting positions, so that multiple pieces will fall down from above on multiple other chess pieces.
- You can also duplicate (**key shortcut Ctrl+D**) the chess pieces (*it will duplicate the whole selected object, including physics behaviours if any*). That s an easy way to have a lot of physical objects that bounce off each other.

After each change, just play and stop physics simulation again.

Make sure that the initial position of all rigid bodies does not make some pair collide with each other right at the start. If the two objects will collide at start, physics engine may (*sometimes quite explosively*) move them away from each other.

This is a sample result:



```
Background1 (TCastleBackground)
Items (TCastleRootTransform)
  Camera1 (TCastleCamera)
  PointLight1 (TCastlePointLight)
  SpotLight (TCastleSpotLight)
  SceneChessBoard1 (TCastleScene)
    Behaviors
      MeshCollider1 (TCastleMeshCollider)
      RigidBody1 (TCastleRigidBody)
  SceneWhiteQueen1 (TCastleScene)
  SceneWhiteRook1 (TCastleScene)
  SceneWhiteRook2 (TCastleScene)
  SceneWhiteKnight1 (TCastleScene)
  SceneWhiteKnight2 (TCastleScene)
    Behaviors
      BoxCollider4 (TCastleBoxCollider)
      RigidBody5 (TCastleRigidBody)
  SceneWhiteKnight3 (TCastleScene)
  SceneWhiteBishop1 (TCastleScene)
  SceneWhiteBishop2 (TCastleScene)
  SceneWhiteKing1 (TCastleScene)
    Behaviors
      BoxCollider2 (TCastleBoxCollider)
      RigidBody3 (TCastleRigidBody)
  SceneWhiteKing2 (TCastleScene)
    Behaviors
      BoxCollider8 (TCastleBoxCollider)
      RigidBody9 (TCastleRigidBody)
  SceneWhiteKing3 (TCastleScene)
```

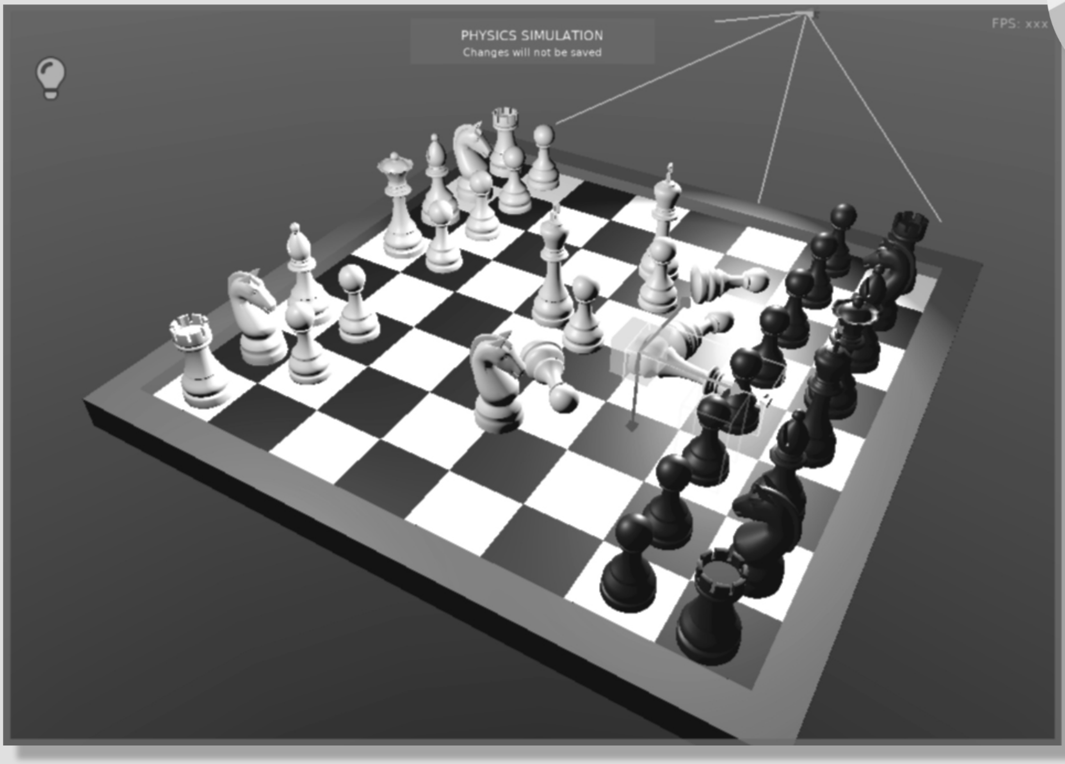
Selected:
SceneWhiteKing3 (TCastleScene)

Basic Layout Events All

AutoAnimation	
AutoAnimation	☑ (True)
Cache	☐ (False)
CastGlobalLigh	☐ (False)
CastShadows	☑ (True)
DefaultAnimati	0
Direction	**-0.97 0 0.25**
Exists	☑ (True)
ExposeTransfor	(TStrings)
Name	SceneWhiteKing3
PreciseCollisior	☐ (False)
ProcessEvents	☐ (False)
RenderLayer	rlParent
RenderOptions	**SceneWhiteKing3.Rende**
Rotation	**0 -1 0 deg(75.52)**
Scale	1 1 1
TimePlaying	☑ (True)
TimePlayingSpe	1
Translation	**-2.22 1.25 -0.56**
Up	0 1 0

```
project
  code
  data
```

Name	Size	Type
code		
data		
bad-chess-project	127 MB	
bad_chess_project_standalone.dpr	1 kB	.dpr
bad_chess_project_standalone.dproj	19 kB	.dproj
bad_chess_project_standalone.lpi	2 kB	.lpi



One last thing remains to learn in this (*first*) part of the article:
how to flick the chess piece?

❶ From **Pascal** code you can use various methods to apply a force on a rigid body. More about this in the next article part.
You can also experiment with the example application `examples/physics/physics_forces/` if you're impatient.

❷ Or you can set a specific `LinearVelocity` on a rigid body component.

We will use the latter approach, as it can be trivially done and tested in the editor.

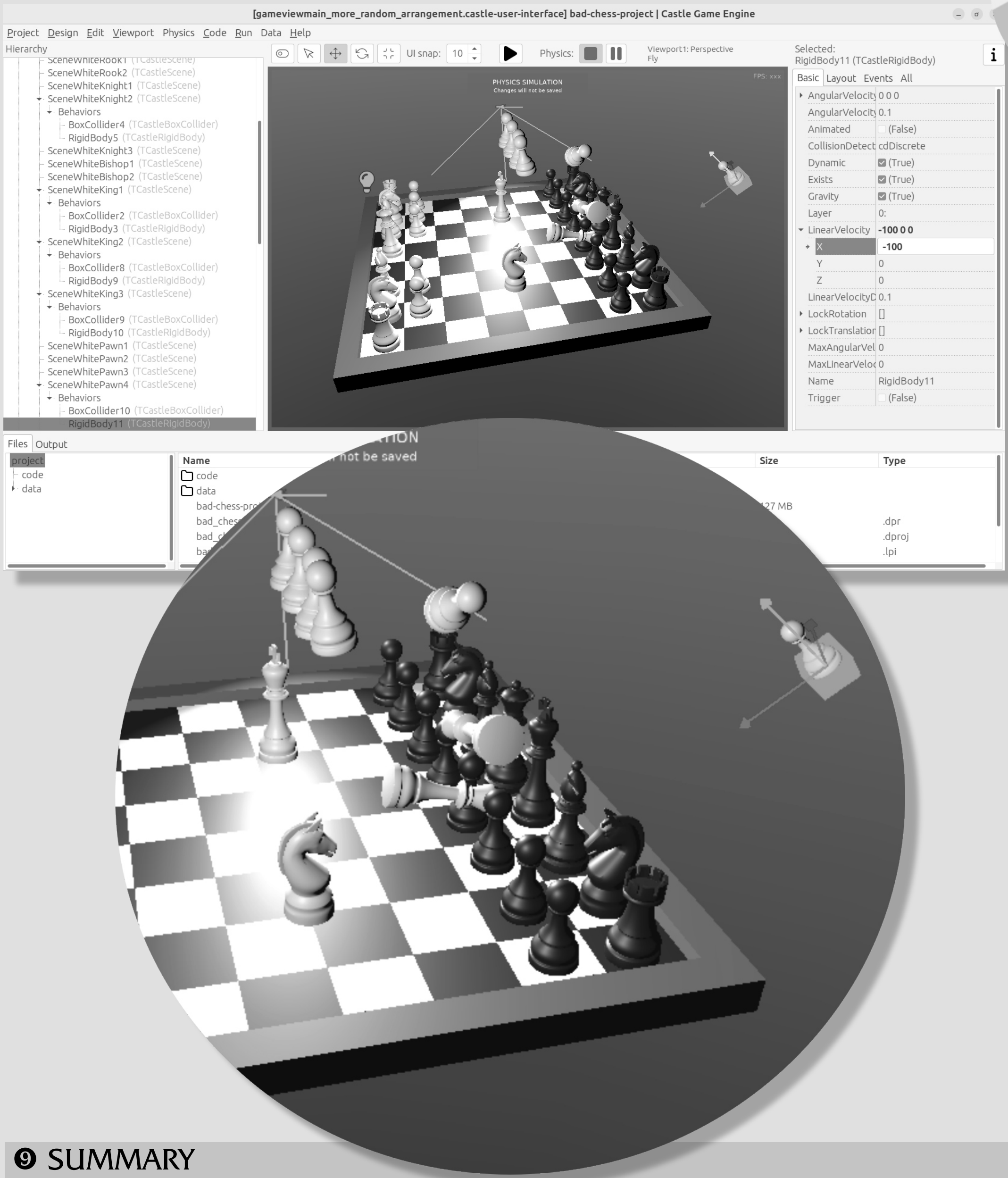
- Select the chess piece. Any chess piece you want to "flick" (*throw across the board*).

- Make sure it has a collider and rigid body components (*if not, add them, as above*).

- Select the `TCastleRigidBody` component of it, and find the `LinearVelocity` property in it.

- Set `LinearVelocity` to any large non-zero vector, like -100 0 0. This means we have a velocity of 100 units per second in the negative X direction.

Run the physics simulation and watch the mayhem.



- SceneWhiteRook1 (TCastleScene)
- SceneWhiteRook2 (TCastleScene)
- SceneWhiteKnight1 (TCastleScene)
- SceneWhiteKnight2 (TCastleScene)
  - Behaviors
    - BoxCollider4 (TCastleBoxCollider)
    - RigidBody5 (TCastleRigidBody)
- SceneWhiteKnight3 (TCastleScene)
- SceneWhiteBishop1 (TCastleScene)
- SceneWhiteBishop2 (TCastleScene)
- SceneWhiteKing1 (TCastleScene)
  - Behaviors
    - BoxCollider2 (TCastleBoxCollider)
    - RigidBody3 (TCastleRigidBody)
- SceneWhiteKing2 (TCastleScene)
  - Behaviors
    - BoxCollider8 (TCastleBoxCollider)
    - RigidBody9 (TCastleRigidBody)
- SceneWhiteKing3 (TCastleScene)
  - Behaviors
    - BoxCollider9 (TCastleBoxCollider)
    - RigidBody10 (TCastleRigidBody)
- SceneWhitePawn1 (TCastleScene)
- SceneWhitePawn2 (TCastleScene)
- SceneWhitePawn3 (TCastleScene)
- SceneWhitePawn4 (TCastleScene)
  - Behaviors
    - BoxCollider10 (TCastleBoxCollider)
    - RigidBody11 (TCastleRigidBody)

Selected:
RigidBody11 (TCastleRigidBody)

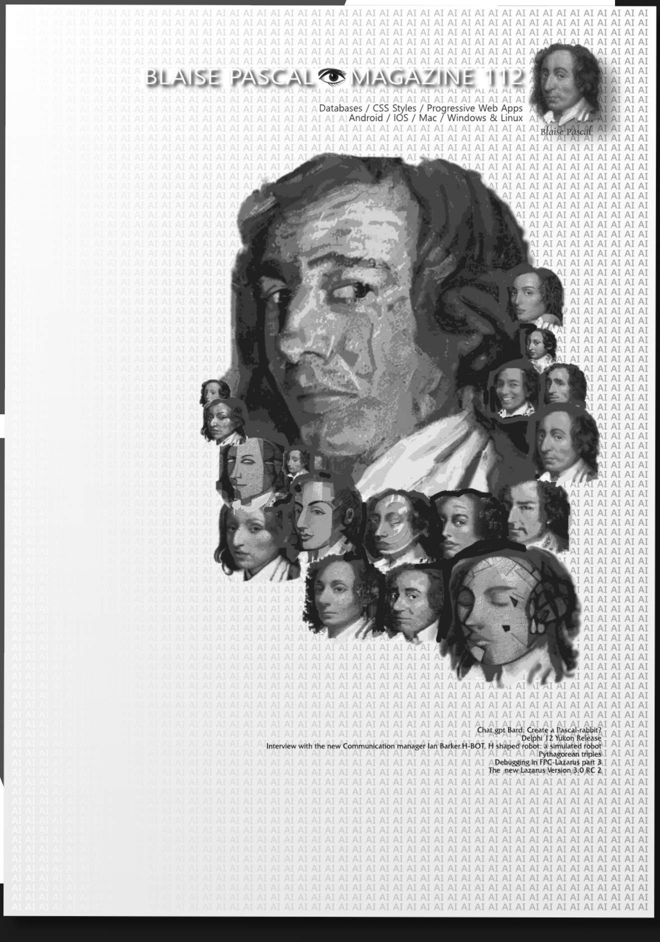
AngularVelocity	0 0 0
AngularVelocity	0.1
Animated	☐ (False)
CollisionDetect	cdDiscrete
Dynamic	☑ (True)
Exists	☑ (True)
Gravity	☑ (True)
Layer	0:
LinearVelocity	-100 0 0
X	-100
Y	0
Z	0
LinearVelocityD	0.1
LockRotation	[]
LockTranslation	[]
MaxAngularVel	0
MaxLinearVeloc	0
Name	RigidBody11
Trigger	☐ (False)

❾ SUMMARY

We have designed a 3D application using **Castle Game Engine** with a bit of physics.
We didn’t yet write any **Pascal** code to do any interactions - **this will be done in the next part of the article.**

If you want to download a ready application, resulting from this, go to `https://github.com/castle-engine/bad-chess` . The subdirectory project of that repository contains the final working demo of this. It will be extended in the next part of the article.
I hope you had fun doing this demo and exploring the possibilities of **Castle Game Engine**.

If you have any questions or feedback about the engine, don’t be shy!
Speak up, ask and share your comments on our forum `https://forum.castle-engine.io` or Discord `https://castle-engine.io/talk.php` .



BLAISE PASCAL MAGAZINE 112

Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

Blaise Pascal

Chat.gpt Bard: Create a Pascal-rabbit?
Delphi 12 Yukon Release
Interview with the new Communication manager Ian Barker.H-BOT, H shaped robot: a simulated robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
The new Lazarus Version 3.0 RC 2

THE LAZARUS DEBUGGER

By Martin Friebe

PART 3: CALLING IT – THE STACK

FROM WHERE WE CAME

The last part of the series has taught us how we can step in and out of functions. We watched what happened in the outer function, and then looked for more details by stepping into another function. This time we will expand our view. We will debug a function, but instead of stepping out to see what happens in the caller, we will get the debugger to show us the caller's data while we are still in the current function. As usual we explore all this by debugging a small sample project.

```pascal
program FindRepeat;
uses Math;

const
 TESTDATA = 'Test a random text. Repeat: a random text';


function EqualSubText(AText: Ansistring; AStart1, AStart2, AMaxLen: Integer): AnsiString;
var
  EqualLen: Integer;
begin
  EqualLen := 0;
  while (AMaxLen > EqualLen) and (AText[AStart1 + EqualLen] = AText[AStart2 + EqualLen]) do
    inc(EqualLen);

  Result := copy(AText, AStart1, EqualLen);
end;

(* DoFindLongestRepeat
   AStart1: Iterates over all potential start positions for the first match of the text
   AMaxSearchLen1: Count of chars up to the start of for the second match
   AStart2: Iterates over all potential start positions for the second match of the text
            Must always be greater the AStart1
                     (or equal, which will be handled by "AMaxSearchLen1 = 0")
    AMaxSearchLen2: Count of chars up to the end of the string
 *)
 function DoFindLongestRepeat(AText, AFound: Ansistring;
    AStart1, AMaxSearchLen1,
    AStart2, AMaxSearchLen2: Integer
  ): Ansistring;
var
  EqualTxt: String;
begin
  Result := AFound;

  EqualTxt := EqualSubText(AText, AStart1, AStart2, Min(AMaxSearchLen1, AMaxSearchLen2));
  if Length(EqualTxt) > Length(Result) then
    Result := EqualTxt;

  if AMaxSearchLen2 > 1 then
    Result := DoFindLongestRepeat(AText, Result,
      // AStart2 increases, so there is one more char available after AStart1
      AStart1,    AMaxSearchLen1 + 1,
      // And there is one char less after AStart2
      AStart2 + 1, AMaxSearchLen2 - 1
    )
  else
  if AStart1 < Length(AText) - 1 then begin
    Result := DoFindLongestRepeat(AText, Result,
      // AStart2 is set equal to AStart1, so AMaxSearchLen1 will be 0
```

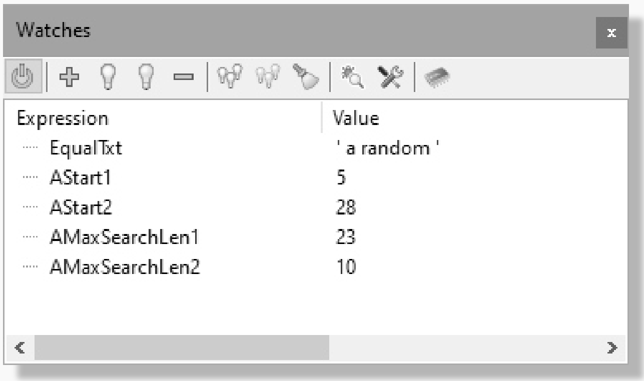
CONTINUATION

```
51.     AStart1 + 1, 0,
52.     // AStart2 will be 1 more than AStart1 was,
53.     // so there will be 1 char less to search after AStart2
54.     AStart1 + 1, AMaxSearchLen1 - 1
55.   );  56.  end;
57. end;
58.
59. function FindLongestRepeat(AText: Ansistring): Ansistring;
60. begin
61.  Result := DoFindLongestRepeat(AText, '',
62.   1, 0,          // AStart1 equals AStart2: There are 0 chars between
63.   1, Length(AText)  // AStart2 has the entire string
64.  );
65. end;
66.
67. begin
68.  writeLn('"' + FindLongestRepeat(TESTDATA) + '"');
69.  readln;
70. end.
```

The code is a recursive example on how to find the longest non-overlapping reoccurring substring. It iterates all combinations of two start-points (**"AStart1"** and **"AStart2"**) and checks for a matching substring on each of them.
For the given test data we expect the result: " **a random text**" (*with leading space*).
When we run it, it will print
" a random "

It somehow misses the last word **"text"**.

As before, we start our debug session by running to a breakpoint.
`Line 38` might be a good candidate to start. It will pause each time a potential result is found. So when the partial result " **a random** " is assigned, we can look at the available data and check if it reveals any clues.
After starting the project with **F9** and hitting the breakpoint, we have a look at the value of "**EqualTxt**". We can do so in the locals or watches window, and we will see it is **"e"**.
As this is not the match we are interested in, we run (**F9**) again.
As we hit the breakpoint for the 2nd time, "**EqualTxt**" will be shown as " **a random** ".
This is the value that got mistakenly printed as the longest repeated match.
We will look at the values that are involved in the call to " **EqualSubText**".



Expression	Value
EqualTxt	' a random '
AStart1	5
AStart2	28
AMaxSearchLen1	23
AMaxSearchLen2	10

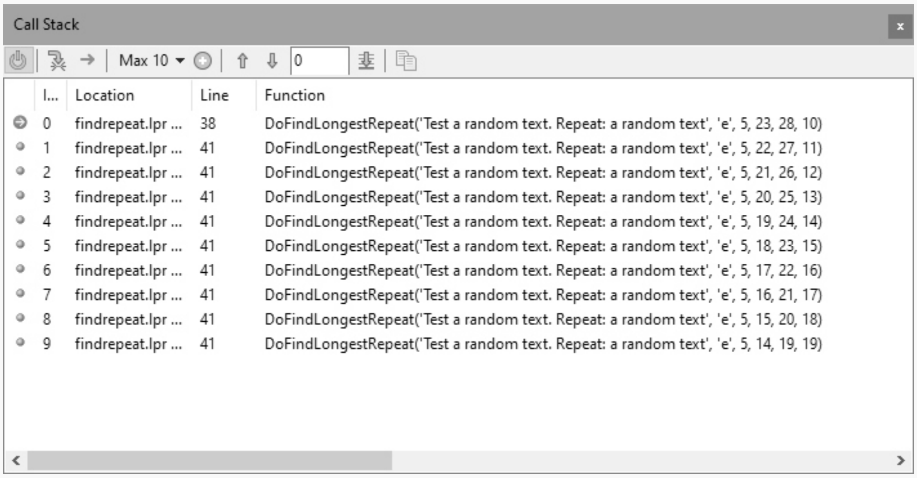
The values for “**AStart1**” and “**AStart2**” look correct.
Checking the 2 max-lengths “**AMaxSearchLen2**” may be worth checking. 10 is only the length to the end of the returned value in “**EqualTxt**”. But we know the “**TESTDATA**” has more text after that. So the value should be larger.

We can check that “TESTDATA” has 41 chars. So if “AStart2” is at 28, then there should be 14 remaining chars. 41 – 28 + 1 (“+ 1” as the last char needs to be included).

“AMaxSearchLen2” has been passed as a parameter by the calling function. To find out more, we need to know what happened in the caller.

THE CALL STACK WINDOW

As announced in the introduction the debugger can help us with this. Ctrl-Alt-S or the menu “View → Debug Windows → Call stack” will open the Call-stack window.



	I...	Location	Line	Function
	0	findrepeat.lpr ...	38	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 23, 28, 10)
	1	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 22, 27, 11)
	2	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 21, 26, 12)
	3	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 20, 25, 13)
	4	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 19, 24, 14)
	5	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 18, 23, 15)
	6	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 17, 22, 16)
	7	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 16, 21, 17)
	8	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 15, 20, 18)
	9	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 14, 19, 19)

Before we continue tracking the wrong value, let’s have a look at the contents of the new window and what information it provides.

The top line is line 38 (*column “Line”*) at which our app is currently paused.

The line below is showing from where the current invocation of “**DoFindLongestRepeat**” was called. As we are in a recursion, the function did call itself. However, the call was invoked from line 41.
Looking at all the columns in the grid.

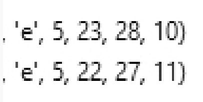
- The first column shows, if the line has a breakpoint. In our case this applies to line 38. But had we had a breakpoint at line 41, it would be shown on the other lines.
- The 2nd column “I...” (Index) is a running number, showing us how many calls we are away from the top. This can be useful, when scrolling through a very long list.
- “Location” and “Line” are the unit (filename) and line-number. If they aren’t known, an address may be shown.
- “Function” is the name of the routine. In case of a method it will be in the “classname.method” notation. This column also contains the values of *the parameters passed. (Please see the note on params and locals in the section “The full stack”)*

We will go into more details later on.

VALUES FROM THE OUTER CALLERS

Before we explore all the features of the callstack we will use the current view to trace the value of “AMaxSearchLen2”.

The argument-values to the call on each line are given in the same order as the parameters are declared in the source. So “**AMaxSearchLen2**” is the last value in the list.



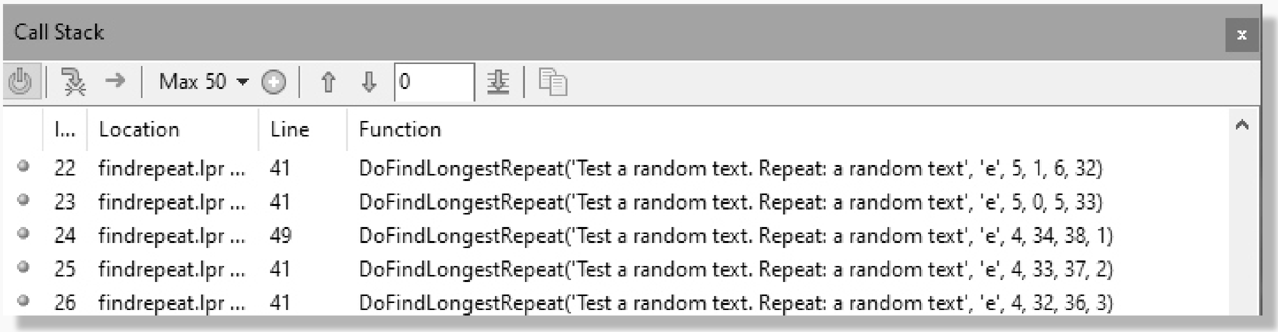
```
. 'e', 5, 23, 28, 10)
. 'e', 5, 22, 27, 11)
```

For the top line – representing the function in which the project is currently paused – we have “**5, 23, 28, 10**” for “**AStart1, AMaxSearchLen1, AStart2, AMaxSearchLen2**”. So as we saw “**AMaxSearchLen2”=10.**

And for the direct caller we have “5, 22, 27, 11”. The caller had checked for repeated text at the position one char earlier (*“AStart2”=27*) and it had up to 11 chars (“ *AMaxSearchLen2*”) to check. Comparing the caller's total of **“27 + 11”** with the current **“28 + 10”**, both functions are the same amount short of the actual full length of the text.

Looking down through the stack on each line “**AStart2**” goes one down, and “**AMaxSearchLen2**” goes one up. However, we only see the top 10 callers, and the relevant information may be further away. We can get more lines, if we press the ⊕ button, or use the “**Max 10**” drop-down. Lets use the “**Max 10**” drop-down and select 50 entries.
(*If we need more we need the* ⊕ *button*)

We scroll down until we find a break in the pattern of +/-1.
This happens for the caller from line 49.



I...	Location	Line	Function
22	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 1, 6, 32)
23	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 5, 0, 5, 33)
24	findrepeat.lpr ...	49	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 4, 34, 38, 1)
25	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 4, 33, 37, 2)
26	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', 'e', 4, 32, 36, 3)

Looking at **index 23** called by **24**, we can see that “**AStart1**” was incremented to 5 (*where the first occurrence of “ a random text” starts*), **and “AStart2” set to start from 5 too. Checking at index 24 we find that “AStart2”+” AMaxSearchLen2” = “38 + 1” = 39.**
Not the full length, but 1 more than **“27 + 11” = 38.**
So during this call we lost 1 char from “**AMaxSearchLen2**”.

Looking at the code

```
Result := DoFindLongestRepeat(AText, Result,
   AStart1 + 1, 0,
   // AStart2 will be 1 more than AStart1 was,
   // so there will be 1 char less to search after AStart2
   AStart1 + 1, AMaxSearchLen1 - 1
 );
```

The code assumes that “**AStart1 + AMaxSearchLen1**” (*ignoring the matching +/-1*) covered the entire string from “**AStart1**” to the end of the string, so that the values can be used for “**AStart2**” and “**AMaxSearchLen2**”.
But actually, they only covered the string up to before “**AStart2**”, which was at the last char of the string. So “**AMaxSearchLen1**” is already one less than the remaining string length. There is no need to subtract 1. The correct code should be:

```
    Result := DoFindLongestRepeat(AText, Result,
      // AStart2 is set equal to AStart1, so AMaxSearchLen1 will be 0
      AStart1 + 1, 0,
      // AStart2 will be 1 more than AStart1 was,
      // so there will be 1 char less to search after AStart2
      AStart1 + 1, AMaxSearchLen1 // no “-1”
     );
```

Re-running the project with this change yields the expected
" a random text"

THE FULL STACK

Now that we solved the issue in the example project, let’s take some more time to explore the stack window. We also should look at a few names often used in this context.

The stack itself has its name from the equally named data structure, also known as a **“LiFo Stack”**. It is a feature many CPU's have. When a function is called, the CPU stores (*pushes*) the current execution address onto the stack, so it can later retrieve it to continue execution in the callers code. Many calls can be nested, and when they return, the addresses are retrieved in reverse order (*Last-in, First-out*).

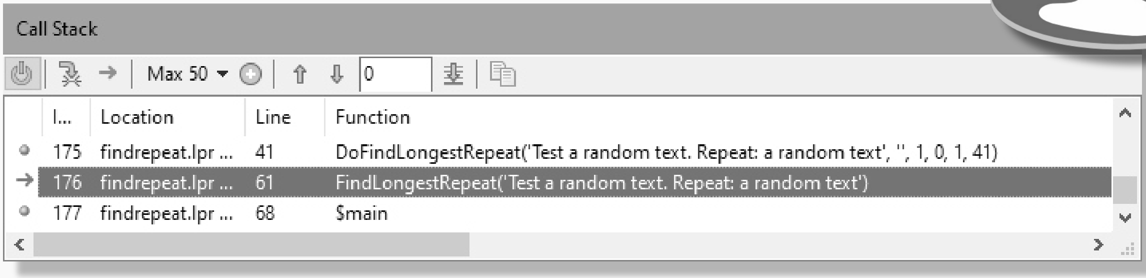
The stack often holds more than just the address for the return. It also holds the values of local variables. This memory on the stack is called a **“stack frame”**.

The name “Frame” or “Stack-frame” is often used to refer to an entry in the stack window.

During the above debug exercise we have seen the values of the function-parameters for each frame listed in the stack. The frame also contains local variables.
The stack window allows us to select any frame as “current”. To do this we select the frame using the mouse or keyboard, and then click the green arrow button: ➜
Once a frame is “current” other debug windows (*like Watches and Locals*) will show their content according to that frame. If we made the frame at index 1 current, then Locals would show the values for that frame, so instead of showing **“AStart2”=28** for the top frame, it would show **“AStart2”=27.**

NOTE: *The parameters and local variables of any frame are shown with their current values. Often that is the value they had at the time the call was made. However, if the called function has changed the local value, then it will show that modified value.*

If we show enough frames, we can see callers other that "**DoFindLongesRepeat**"



I...	Location	Line	Function
175	findrepeat.lpr ...	41	DoFindLongestRepeat('Test a random text. Repeat: a random text', '', 1, 0, 1, 41)
176	findrepeat.lpr ...	61	FindLongestRepeat('Test a random text. Repeat: a random text')
177	findrepeat.lpr ...	68	$main

Making "**FindLongestRepeat**" the current frame, and we can see that the locals window no longer shows `AStart.../AMaxSearchLen` variables. The locals show "**AText**" instead, which is the parameter passed to "**FindLongestRepeat**".
We also see that we can trace back all the way to the program's "`begin...end.`" block shown as "`$main`". And the index tells us, that our recursion is a 175 calls deep at the time of hitting the breakpoint.

Sometimes stack traces are much deeper than that, and in that case using the ⊕ button to reach the bottom of the stack can be tedious.
In this case the blue ⇧ ⇩ up/down buttons can help to navigate quickly to the top and bottom. And the edit field with the button can be used to enter any index and show frames starting from it.
On the topic of navigation, the stack window also allows us to navigate in the source code. Double clicking any line in the stack (*or using the button*) will bring up the code in the source editor.
Of course only, if the **stackframe** has a source-file and line-number.
The button will copy all entries to the clipboard. And the power button will freeze the currently shown entries. When power is off, the stack window will not update when you step/run the application. In case you want to keep the current frame list as a reminder or something like that.

SUMMARY
The callstack can be used to inspect locals from any caller. It can also show us who called the current function.

- Open the Stack Window:
  - **Ctrl-Alt-S**
  - **Menu: View → Debug windows → Call stack**
- Select a frame as "current"
  - →
- Increase amount of shown frames
  - ⇧ ⇩
  - "Max 10" Drop-down

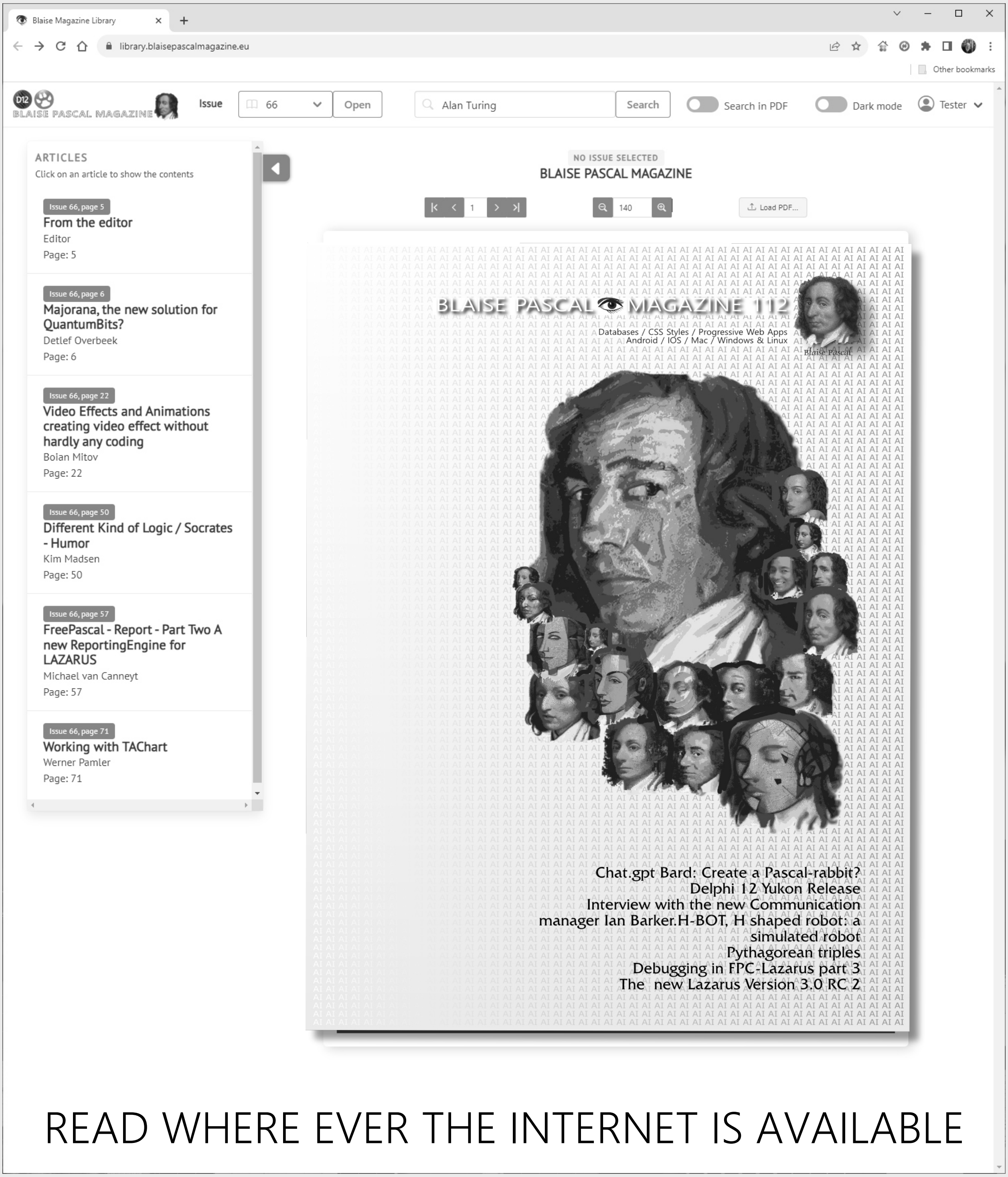
In the next article: Part 4: **TAKING A LOOK – WATCHES**

* * *

THE NEW INTERNET
BLAISE PASCAL LIBRARY 2023

https://library.blaisepascalmagazine.eu/

JUST OPEN ANY BROWSER (CHROME, SAFARI, EDGE, FIREFOX, OPERA, DUCKDUCKGO) AND LOGIN: YOU WILL HAVE ALL ISSUES AVAILABLE - 6500 PAGES. FOR ALL ISSUES STARTING AT NR1 UP TO THE LATEST ITEM. YOU NEED A VALID SUBSCRIPTION ONLY € 50,00 - VALID THROUGH ONE YEAR



ARTICLES

Click on an article to show the contents

Issue 66, page 5
From the editor
Editor
Page: 5

Issue 66, page 6
Majorana, the new solution for QuantumBits?
Detlef Overbeek
Page: 6

Issue 66, page 22
Video Effects and Animations creating video effect without hardly any coding
Boian Mitov
Page: 22

Issue 66, page 50
Different Kind of Logic / Socrates - Humor
Kim Madsen
Page: 50

Issue 66, page 57
FreePascal - Report - Part Two A new ReportingEngine for LAZARUS
Michael van Canneyt
Page: 57

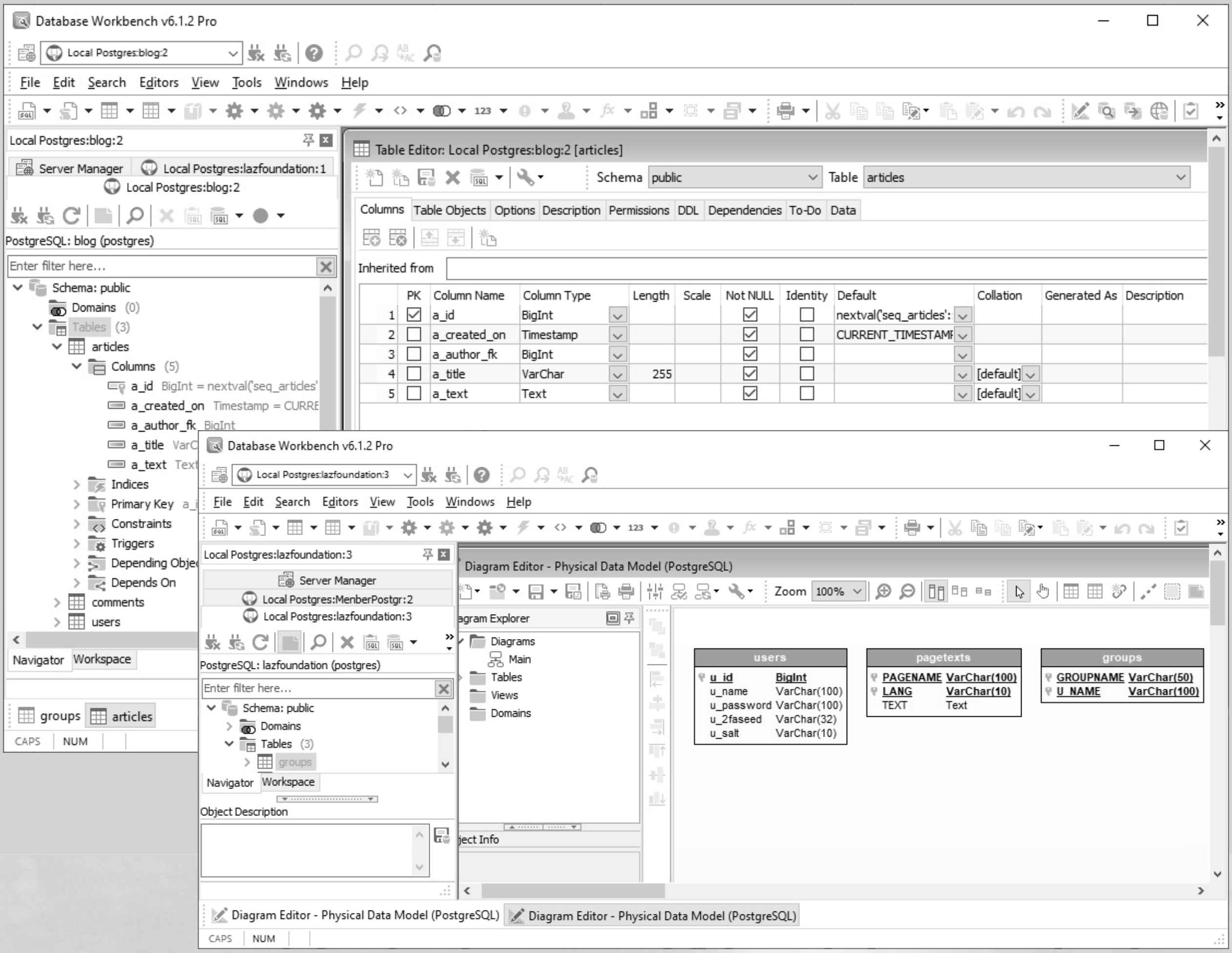
Issue 66, page 71
Working with TAChart
Werner Pamler
Page: 71

NO ISSUE SELECTED

BLAISE PASCAL MAGAZINE

Chat.gpt Bard: Create a Pascal-rabbit?
Delphi 12 Yukon Release
Interview with the new Communication manager Ian Barker.H-BOT, H shaped robot: a simulated robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
The new Lazarus Version 3.0 RC 2

READ WHERE EVER THE INTERNET IS AVAILABLE

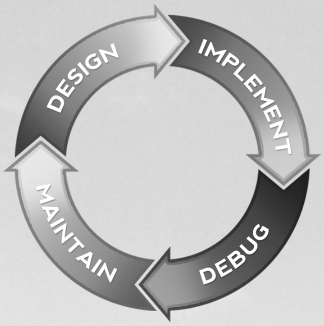


	PK	Column Name	Column Type	Length	Scale	Not NULL	Identity	Default	Collation	Generated As	Description
1	☑	a_id	BigInt			☑	☐	nextval('seq_articles':			
2	☐	a_created_on	Timestamp			☑	☐	CURRENT_TIMESTAMF			
3	☐	a_author_fk	BigInt			☑	☐				
4	☐	a_title	VarChar	255		☑	☐		[default]		
5	☐	a_text	Text			☑	☐		[default]		

users	
u_id	BigInt
u_name	VarChar(100)
u_password	VarChar(100)
u_2faseed	VarChar(32)
u_salt	VarChar(10)

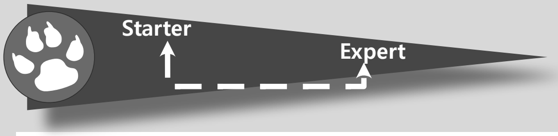
pagetexts	
PAGENAME	VarChar(100)
LANG	VarChar(10)
TEXT	Text

groups	
GROUPNAME	VarChar(50)
U_NAME	VarChar(100)



EXECUTING PROGRAMS ON THE SERVER IN PAS2JS

By Michael Van Canneyt



ABSTRACT
In this article we show how to give the user of a browser-based program feedback from long-running processes on the server, using 2 components: one in **PAS2JS**, one in **Free Pascal/Lazarus**.

❶ INTRODUCTION

When using a web-based program, not everything can be done in the browser.
Often,tasks are executed through some **RPC** (**Remote Procedure Call**) mechanism on the webserver. This can be a simple task such as executing an SQL statement on a database and returning a result. Or it can be a more complicated and time-consuming task such as making a backup of a database, indexing PDF files, compiling a software project and running a test suite, or even installing software on the server. Ideally, the output of these remote programs should also be presented to the user.

To keep programs scalable, these tasks should be short-lived. A return time of 1second for a HTTP request is already a long time, so executing a time-consuming task and waiting for the return using a single HTTP request is not a good idea:
the HTTP server is occupied with the request, the browser or any proxy servers between the HTTP server and the browser may decide to time-out your request.

Much better is to start the process using a HTTP request, and use a mechanism to poll the status of the executed process. In this article we present one such mechanism.

❷ ARCHITECTURE

The solution we present here consists of 2 components. One component which is used on the server, and which can be used to start a process, capture its output and poll for the status of the process. The other component takes care of the polling process on the client.

These components are ignorant of the communication mechanism between browser and server, this means that they do not implement the actual **RPC** calls used to start the process: There are many possible mechanisms, and some may be more suitable for your purpose than others.

The components are called `TProcessCapture` for the server part and `TProcessCapturePoller` for the client (**PAS2JS**) part. The server part takes care of executing a program and redirecting the output to a file, the client part implements the polling mechanism and some callbacks to handle the actual server calls and the result. We'll demonstrate both components with a simple set of programs:

- A test program to be executed.
  It is used for demonstration purposes only.
- A HTTP server program that allows to serve HTML files and that offers an
- RPC mechanism to start the test program and handle status requests. A Simple **PAS2JS** program that will run in the browser and which will remotely execute the test program. It will show the output of the test program in the browser.

We'll start with the test program.

❸ THE TEST PROGRAM

To demonstrate the workings, the test program needs to do 3 things:

- ❶ It must run for some time, several seconds at least.
  This is done with a simple loop and a call to sleep.
- ❷ needs to show that it receives command-line arguments:
  we will simply output the program parameters.
- ❸ It must demonstrate that it is run in a specific directory.
  We’ll just print the working directory.
- ❹ It needs to produce some output.

All this is easily accomplished with a trivial program:

```
uses sysutils;

var
  i : integer;
  D : TDateTime;
begin
  Writeln(’Current dir: ’, GetCurrentDir);
  Write(’Args:’);
  For I:=1 to ParamCount do
    Write(’ ’,ParamStr(i));
  Writeln();
  D:=Now;
  For I:=1 to 150 do
    begin
    Sleep(100);
    Writeln(’Tick ’,i);
    Flush(output);
    end;
  Writeln(SecondsBetween(Now,D), ’ seconds elapsed’);
  flush(output);
end;
```

The only noteworthy thing about this program is that it flushes standard output after writing a line: By default, **Free Pascal** buffers output of writeln statements if it detects that it is not writing to a console. Since our program will be run with the output redirected, the buffering will be activated, and so, in order to send the output faster to the browser, we flush standard output manually.

❹ THE SERVER COMPONENT

Before explaining the server component, it is a good idea to explain why a new component is needed. After all, **Free Pascal** ships since ages with the `TProcess` component, which can be used to start a process and read its output using a stream. So why not simply use that ? This component is not really suitable for our task, for several reasons:

- A web server process (`e.g. cgi, fastcgi`) can be ended before the process has finished. All information about the executed process would be lost.
- The component cannot be used to redirect output to a file. It would require reading all data from the file in a separate thread, save it somewhere etc. This complicates matters considerably, and if the HTTP program ends, all further input/output would stop. Similarly, no input file can be specified, it would require similar handling as the output file.
- Item since the TProcess component is confined to a single process, there is no way to scale your web application.

In essence, the `TProcess` component is stateful, and we need a stateless component in order to work in a web environment.
So, a new component is needed.
The server component `TProcessCapture` has the following declaration:

```
TProcessCapture = Class(TComponent)
Public
   Function Execute(Exe : String; Args: Array of string) : string;
   Function Execute(Exe : String; Args: TStrings) : string;
   Function CleanupProcess(Const AProcess : String) : Boolean;
   Function GetOutputFile(Const AProcess : String) : String;
   Function GetPidFile(Const AProcess : String) : String;
   Function GetStatusFile(Const AProcess : String) : String;
   Function GetProcessID(Const AProcess : String) : Integer;
   Function IsProcessRunning(Const AProcess : String) : Boolean;
   Function GetProcessExitStatus(Const AProcess : String) : Integer;
   Function GetProcessOutput(Const AProcess : String; Var AOffSet : Integer) : RawByteString;
Published
   Property LogDir : String Read FLogDir Write FLogDir;
   Property InputFile : String Read FInputFile Write FInputFile;
   Property Working-Dir : String Read FWorkingDir Write FWorkingDir;
   Property OutputCodePage : TSystemCodePage Read FOutputCodePage Write FOutputCodePage;
end;
```

The main methods are almost self-explanatory:

- **`Execute`** - executes the program Exe, passing it the arguments Args, which can be given as an array of strings, or a stringlist. The return of this function is a process identifier.
- **`CleanupProcess`** - will clean up the output and status files for the process identified by `AProcess`. You should call this only after the process has exited.
- **`IsProcessRunning`** - returns True if the process identified by `AProcess` is still running.
- **`GetProcessExitStatus`** – returns the exit status of the process identified by `AProcess`. If the process is still running, -1 is returned.
- **`GetProcessOutput`** - returns the output of the process identified by `AProcess`, starting at byte offset `AOffSet` (*zero based*) till the end of available output. There are some auxiliary methods that you do not need under ordinary circumstances:
- **`GetOutputFile`** - returns the name of the output file associated with the process `aProcess`.
- **`GetPidFile`** - returns the name of the process ID file associated with the process `aProcess`. This file will be created as soon as the process starts.
- **`GetStatusFile`** - returns the name of the status file associated with the process `aProcess`. This file will only exist after the program has exited.
- **`GetProcessID`** - returns the process ID of the process `AProcess`.

Lastly, there are some properties:

- **`LogDir`** - the directory where all log and status files are created. The directory will be created if it does not exist.
- **`InputFile`** - a file with prepared input for the process. NOTE that this does not allow you to interact with the process. This property is only used when starting the program.
- **`WorkingDir`** – The working directory for the started program. This property is only used when starting the program.
- **`OutputCodePage`** - The codepage in which the program writes its output.

To work with this component, you will typically perform the following steps:

❶ Set appropriate values for `LogDir`, `InputFile`, `WorkingDir` and `OutputCodePage`. They contain sensible defaults, but it is better to be explicit.
❷ Start the program using the `Execute` method, and save the resulting `ProcessID` string.
❸ Initialize an offset variable to zero.
❹ Check if the process is still running with `IsProcessRunning`, passing it `ProcessID`.
❺ Get the output of the process using `GetProcessOutput`, passing it `ProcessID` and the current offset. Update the offset.
❻ Repeat the last 2 steps till the program exits.

It should be noted that you can free the `TProcessCapture` after every step and recreate it before performing a call: it is stateless. This is necessary if the component is to work in a web environment where the different steps will be performed as part of different HTTP requests: the steps may be performed by different instances of the application server.
To work correctly, the `LogDir` and `OutputCodePage` properties must be set to the same values between invocations.

It also means that the same component can be used to control different processes. Although this is not recommended if you use threads: the component is not re-entrant.

To do its work, the `TProcessCapture` component executes a small helper program called `taskhelper`: this program does the work of launching the actual program that needs to be executed with redirected in and output. It also takes care of registering the exit status of the program.
On **Unix** platforms, it is possible to do without this program, but on **Windows**, the mechanism to start a new process `CreateProcess` necessitates the use of an extra program.
To make the behavior across platforms consistent, the `taskhelper` program is used everywhere.Its sources are distributed with the trunk version of FPC, but the source has been included in the sources of this article.

❺ THE SERVER PROGRAM

To demonstrate the working of the component, we'll make a small HTTP server that executes the test program when it receives a `StartProcess` command from the browser through **JSON-RPC**, and which has a `GetStatus` command to get the status of the process.

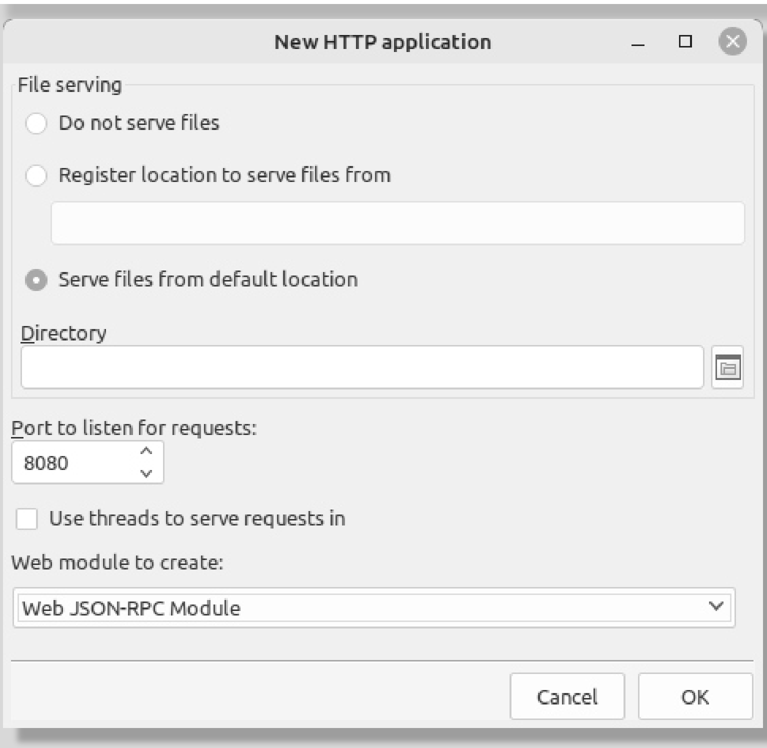
The process will also serve the files for the client application.
To do this, in the '**New project**' dialog we select '**HTTP Server application**', and in the wizard that is shown we select '**Server files from default location**' and under '**Web module to create**' we select '**Web JSON-RPC Module**', as shown in *figure 1 on page 5* of the article.
In the next dialog which creates the module to **JSON-RPC Module**, we only need to register the web module
(*we have only 1 module in the server application*),
(*see figure 2 on page 6*) and we'll use the `/RPC URL` path to serve **JSON-RPC** requests from.
Once that is done, we need 2 `JSONRPCHandler` components from the `FPWeb` tab in the component palette, one for each request:

`StartProcess` - this call takes 2 arguments: 2 strings, which we must define in the Params property of the component. We'll give them the names A and B. The call will return the process identifier to the client application.

`GetStatus` this call also takes 2 arguments: a string (*the ProcessID*), and an **Int64** number (*an offset*), we also define them in the Params property. The call will return the process exit code (-1 if the process is still running), the available output starting at the given offset identifier to the client application. It also returns the new offset.

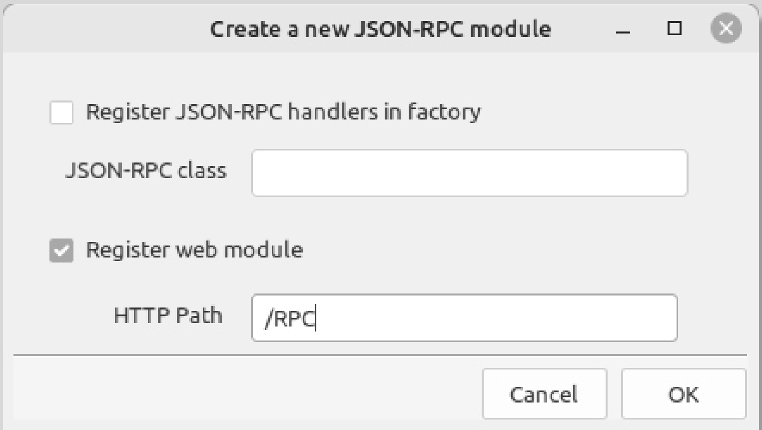
These 2 **RPC** calls are the API we expose to the browser to control our process.
The actual work is done by the `TProcessCapture` component.

The `TProcessCapture` component is not (yet) on the component palette of Lazarus, so we create it in code in the `OnCreate` handler of the datamodule, and destroy it in the `OnDestroy` handler:



File serving
○ Do not serve files
○ Register location to serve files from
● Serve files from default location
Directory
Port to listen for requests: 8080
☐ Use threads to serve requests in
Web module to create: Web JSON-RPC Module

Figure 1: The start of the server application



☐ Register JSON-RPC handlers in factory
JSON-RPC class
☑ Register web module
HTTP Path /RPC

Figure 2: Creating the JSON-RPC web module

```
procedure Tprocesscontrol.DataModuleCreate(Sender: TObject);
begin
  Capture:=TProcessCapture.Create(Self);
end;

procedure Tprocesscontrol.DataModuleDestroy(Sender: TObject);
begin
  FreeAndNil(Capture);
end;
```

The latter is strictly speaking not necessary since the component is owned by the datamodule and will be destroyed when the datamodule is destroyed, but for clarity we destroy it manually anyway.
In the `OnExecute` event of the `StartProcess` handler, we collect the 2 arguments A and B and start the test program:

```
const
  LongProcess = 'longprocess' {$ifdef windows} + '.exe' {$endif} ;

var
  arr : TJSONArray absolute Params;
  a, b, Exe, PID : string;
begin
  Res:=Nil;
  a:=Arr.Strings[0];
  b:=Arr.Strings[1];
  Exe:=ExtractFilePath(ParamStr(0))+longprocess;
  PID:=Capture.Execute(Exe,[a,b]);
  Res:=TJSONString.Create(PID);
```

As you can see in this code, we use the `Execute` method of the `TProcessCapture` class to start the process.
For the `GetStatus` call, the code is a little longer, but not so much.
The code starts by getting the arguments, and checking the whether the process is still running. If the process is no longer running, then the exit status is retrieved.

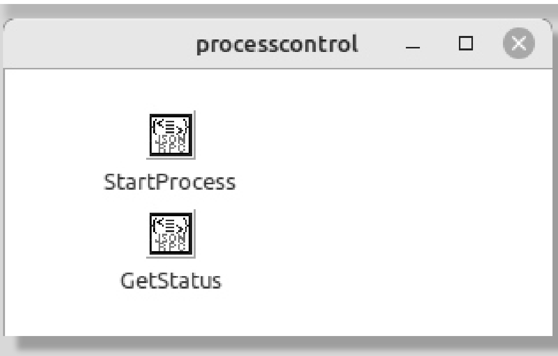


Figure 3: The finished JSON-RPC web module

```
procedure Tprocesscontrol.GetStatusExecute(Sender: TObject;
   const Params: TJSONData; out Res: TJSONData);
var
   arr : TJSONArray absolute Params;
   PID,aOutput : string;
   Offset,Status : Integer;
begin
   Res:=Nil;
   PID:=Arr.Strings[0];
   OffSet:=Arr.Int64s[1];
   if Capture.IsProcessRunning(PID) then
     Status:=-1
   else
     Status:=Capture.GetProcessExitStatus(PID);
   aOutput:=Capture.GetProcessOutput(PID,Offset);
   Res:=TJSONObject.Create([’status’,Status,’output’,aOutput,’offset’,offset]);
end;
```

Regardless of whether the process was still running or not, finally the available output is retrieved and all 3 elements (*status, output, new offset*) are returned to the client in a **JSON** object.

The data module will look like figure 3 on page 6 of this article.

Before the program can be used, there are two last things to be done when using the release version of **FPC** on **Linux**. The **HTTP** connection on which requests arrive is passed to the task helper, and as a consequence the connection is not closed
when the `StartProcess` call returns, causing the browser to wait till the process exits. This of course defeats the purpose of the whole exercise.
To remedy this, we must set the Close-On-Exec flag on the socket handle.
This can be done easily by handling the `OnAllowConnect` handler of the **HTTP** server.

To do so, we add the following to the project file:

```
THTTPApplication = Class(fphttpapp.THTTPApplication)
  constructor Create(aOwner : TComponent); override;
private
  procedure DoConnect(Sender: TObject; ASocket: Longint; var Allow: Boolean);
end;

{ THTTPApplication }

constructor THTTPApplication.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  OnAllowConnect:=@DoConnect;
end;

procedure THTTPApplication.DoConnect(Sender: TObject; ASocket: Longint;
                                             var Allow: Boolean);
{$IFDEF UNIX}
const
FD_CLOEXEC = 1;
 {$ENDIF}
begin
  {$IFDEF UNIX}
FpFcntl(aSocket, F_SETFD, FD_CLOEXEC);
  {$ENDIF}
   Allow:=True;
end;
```

Lastly, to serve the files of the client program, we set the base directory for the file serving module to the directory with the client program files:

```
Function GetBaseDir : String;
begin
  Result:=ExtractFilePath(ParamStr(0));
  Result:=Result+’..’+PathDelim+’client’;
  Result:=ExpandFileName(Result);
end;
```

(*this code assumes there are 2 directories: one for the server, one for the client.*) Finally, we load all known mime types, and create our own HTTP application:

```
Var
 Application:THTTPApplication;
begin
 MimeTypes.LoadKnownTypes;
 TSimpleFileModule.BaseDir:=GetBaseDir;
 TSimpleFileModule.RegisterDefaultRoute;
 Application:=THTTPApplication.Create(Nil);
 Application.Title:=’Process server’;
 Application.Port:=8060;
 Application.Initialize;
 Application.Run;
 Application.Free;
end;
```

NOTE that we set the HTTP port to port 8060

6 THE BROWSER CLIENT-SIDE COMPONENT

In the browser the `TProcessCapturePoller` component is used to help working with the `TProcessCapture` component on the server. It does not start the actual process, it just takes care of polling the server for the status of the started process, and triggers a series of events based on results. It also handles the state of the output offset parameter. There are properties to control how often and how long the polling mechanism must try, and how many errors can be tolerated before the polling is abandoned.

To be agnostic of the actual **RPC** mechanism used, the actual poll is also achieved using an event. It is the responsibility of the programmer to implement this event, and to use the `ReportProgress` mechanism to communicate the server results to the component.

This component has the following declaration:

```
 Type
  TProcessStatus = (psRunning, // Process still running
                    psExited, // Process has exited
                    psError // Too many errors
                    );

  TOnGetProcessStatusEvent =
          Procedure (Sender : TObject; aProcessID : String; aOffset : NativeInt)
  TOnProcessDoneEvent =
          Procedure (Sender : TObject; aStatus : TProcessStatus; aExitCode : Integer)
  TOnProcessOutputEvent =
          Procedure (Sender : TObject; aOutput : String) of object;
  TOnStatusFailEvent =
          Procedure (Sender : TObject; aError : String) of object;

TProcessCapturePoller = class(TComponent)
```

```
Public
  Procedure Start;
  Procedure Cancel;
  Procedure ReportProgress(aStatus : TProcessStatus;
                           aOutput : String;
                           aExitCode : Integer;
                           aOffSet : NativeInt);
  Procedure ReportProgressFail(const aMessage : string);
  Property Canceled : Boolean ;
  Property FailCount : Integer;
  Property StatusCheckCount : Integer;
  Property OutputOffset : NativeInt;
Published
  Property ProcessID : String;
  Property OnGetProcessStatus : TOnGetProcessStatusEvent;
  Property OnProcessDone : TOnProcessDoneEvent;
  Property OnProcessOutput : TOnProcessOutputEvent;
  Property OnStatusFail : TOnStatusFailEvent;
  Property LinebasedOutput : Boolean;
  Property PollInterval : Integer;
  Property MaxFailCount : Integer;
  Property MaxCheckCount : Integer;
end;
```

The methods perform the following tasks

- **`Start`** – this starts the polling process.
- **`Cancel`** – this cancels the polling process.
- **`ReportProgress`** - this method must be used when the `OnGetProcessStatus` event handler received the status of the process from the server. The `aStatus` parameter is one of the available statuses, `aOutput` is the output of the process.
  Parameter `aExitCode` is the exit code (*in case status is* `psExited`) and `aOffset` is the new offset (`as reported by the server`).
- **`ReportProgressFail`** - this method must be used when the server call to get the process status failed. The `aMessage` status parameter can be used to indicate what exactly failed.

The following events can be handled:

- **`OnGetProcessStatus`** - This is the only event that must be implemented. It is triggered at regular intervals, when the poller needs to inquire the status of the server process. The poller will pass the process ID and current output offset to the event, so the user does not need to track the state of these parameters.
- **`OnProcessDone`** - This is called when the process has exited or the polling was canceled. It reports the status (`psError` *in case of error*) and the exit code of the process.
- **`OnProcessOutput`** – This is called when output of the process was received: The `aOutput` parameter contains the reported output. This event will be called multiple times.
- **`OnStatusFail`** - This is called when the `ReportProgressFail` was called to signal a failure of the call to get the status of the process. It can be called multiple times, depending on the value of `MaxFailCount`.

The behavior of the component is controlled by the following properties:

- **`LinebasedOutput`** - If set to `True` the component will split the received output in lines, and will call `OnProcessOutput` for each line instead of reporting the whole received output in one call (*if set to False*)

- **`PollInterval`** the time period (*in milliseconds*) after which `OnGetProcessStatus` event is triggered. Default is 500ms. **NOTE** that the event is only retriggered after the result (success or failure) of the previous event has been reported. This is done in order to avoid overlapping getstatus calls.
- **`MaxFailCount`** The maximum number of failures that may be reported before polling is abandoned. Default is 1.
- **`MaxCheckCount`** The maximum number of times the component will poll before reporting a timeout.

Finally, the following properties can be used to get some information about the polling process:

- **`Canceled`** The polling process was canceled.
- **`FailCount`** – The number of failures since the polling was started.
- **`StatusCheck - Count`** The number of times the status will still be checked.
- **`OutputOffset`** – The current output offset.

It may seem strange to have the `OnProcessDone, OnStatusFail` and `OnProcessOutput` events if the fetching of the process status must be implemented in an event: surely the event handler can display the output, decide when the process has ended etc.

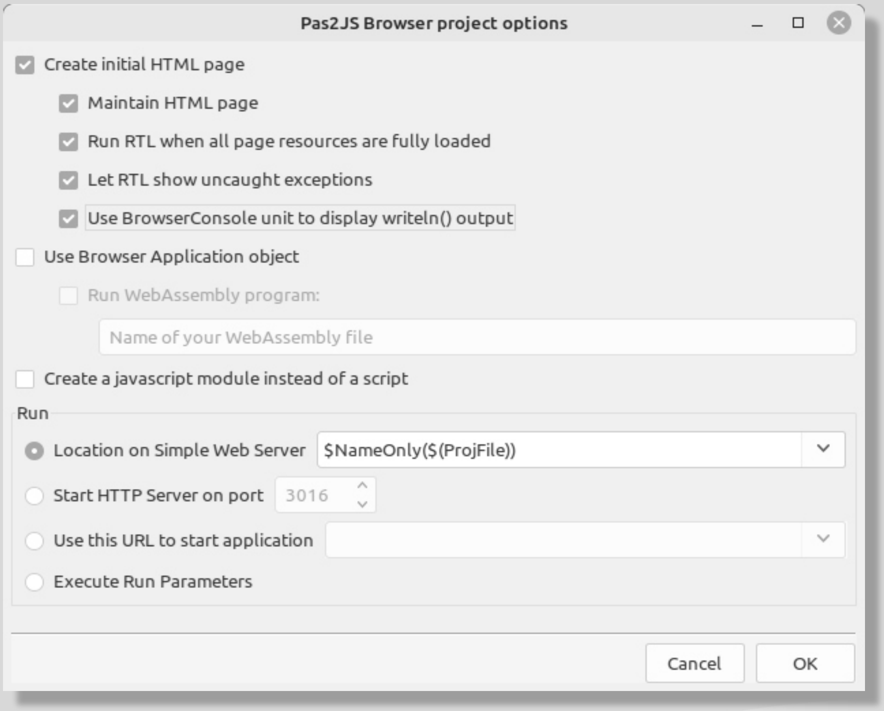
The reason is twofold: first of all, the state logic for the output can be handled by the component, but more importantly: by having these events available, the component can easily be used as a parent for descendents that incorporate the polling **RPC** mechanism in the component. (*as will be demonstrated below*).

❼ THE CLIENT PROGRAM

Armed with this component, we can now start the client side program. In the 'Project - New project' dialog we select the 'Web browser program' item, and enter the correct settings, as shown in figure 4 on page 13. The html file is best saved as index.html.

The HTML needs 5 elements:

❶ A button to start the process.
❷ A button to cancel the polling process.
❸ An edit for parameter A for the started program.
❹ An edit for parameter B for the started program.
❺ An HTML element in which the output of the program will be shown. We will use the browser console unit output mechanism for this: a simple `Writeln` statement will result in the appending of the output to this element.



Pas2JS Browser project options

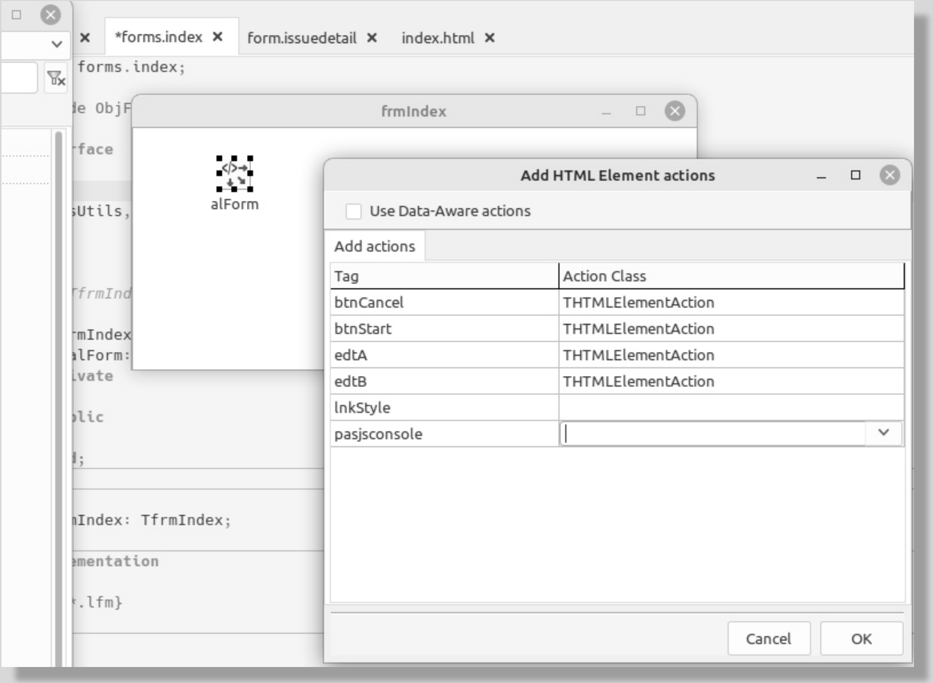
☑ Create initial HTML page
  ☑ Maintain HTML page
  ☑ Run RTL when all page resources are fully loaded
  ☑ Let RTL show uncaught exceptions
  ☑ Use BrowserConsole unit to display writeln() output
☐ Use Browser Application object
  ☐ Run WebAssembly program:
    Name of your WebAssembly file
☐ Create a javascript module instead of a script

Run
● Location on Simple Web Server: $NameOnly($(ProjFile))
○ Start HTTP Server on port: 3016
○ Use this URL to start application
○ Execute Run Parameters

Figure 4: Creating the client program

The following simple HTML (*using Bulma CSS*) will do the job just fine:

```
<h3 class="title is-3">Process output demo</h3>
<div class="box">
   <h4 class="title is-4">Start parameters</h4>
   <div class="field">
     <label class="label">Argument A</label>
     <div class="control">
       <input id="edtA" type="text" class="input"
               placeholder="Enter argument A">
     </div>
   </div>
<div class="field">
   <label class="label">Argument B</label>
   <div class="control">
     <input id="edtB" type="text" class="input"
       placeholder="Enter argument B">
   </div>
</div>
<div class="field is-grouped">
   <div class="control">
     <button id="btnStart" class="button is-primary">
       Start process
     </button>
   </div>
   <div class="control">
       <button id="btnCancel" class="button is-warning is-light">
          Cancel
       </button>
     </div>
   </div>
</div>
<div class="box">
   <h4 class="title is-4">Process output</h4>
   <div id="pasjsconsole">
   </div>
</div>
```



*forms.index × form.issuedetail × index.html ×

frmIndex

alForm

Add HTML Element actions

☐ Use Data-Aware actions

Add actions

Tag	Action Class
btnCancel	THTMLElementAction
btnStart	THTMLElementAction
edtA	THTMLElementAction
edtB	THTMLElementAction
lnkStyle	
pasjsconsole	

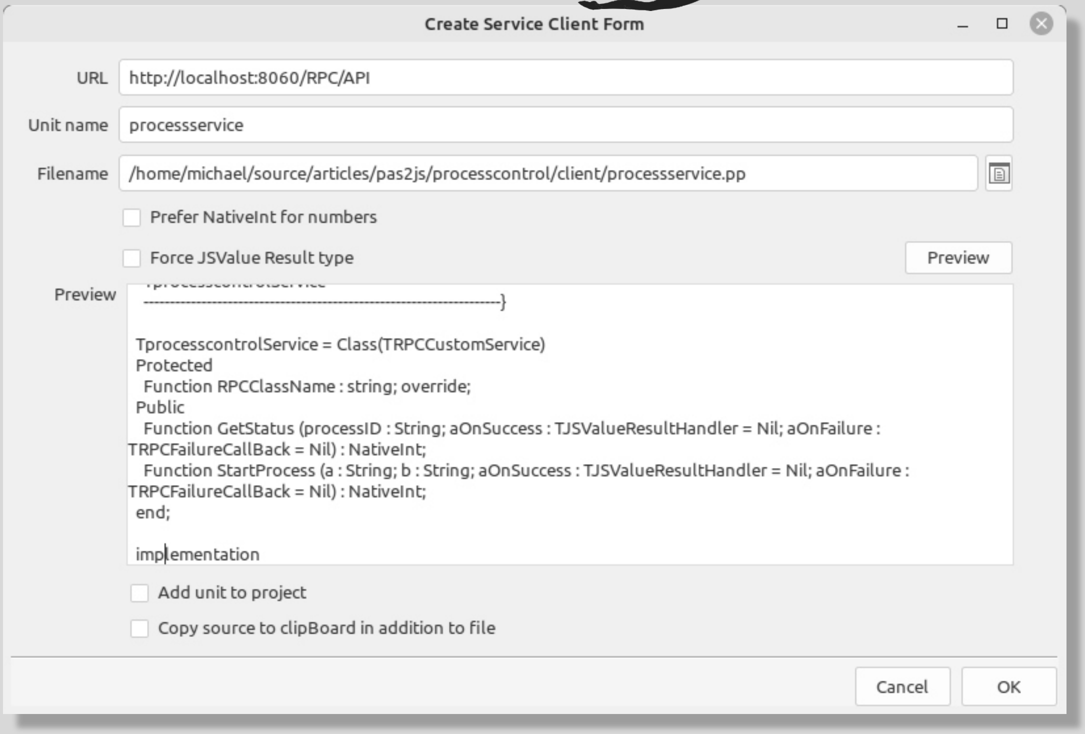
Cancel OK

Figure 5: The HTML form tags

To interact with this HTML, we first create a HTML Fragment module using the '**File - new**' dialog. We name it '**frmIndex**' and set the '**UseProjectHTML**' property to `True`. On this module, we drop a `THTMLElementActionList` component from the component palette. Using the component context menu '**Create actions for HTML tags**', we can create actions for all tags in the above HTML, as shown in figure 5 on page 12 of the article.
We need a `TPas2jsRPCClient` from the **Pas2JS** tab in the component palette: this component will handle the **RPC requests**, and we'll name it **RPC** for short. The component can only do its work correctly if it knows where the server is: We need to enter the URL property. As shown in an earlier article, we can now generate a service proxy: this is a class which has correct method definitions, reflecting the methods defined in our **RPC** server. Calling these service methods will actually execute the methods on the server. Right-clicking on the RPC component and selecting '**Create Service Client component**' shows the service generation dialog as shown in figure 6 on page 13 of the article. We name the unit '`processservice`' and tell the IDE to add it to the project.

Now we can start coding the application. We will create the `TProcessCapturePoller` and service client in the `OnCreate` event of our index form module:

```
procedure TfrmIndex.DataModuleCreate(Sender: TObject);
begin
  Service:=TprocesscontrolService.Create(Self);
  Service.RPCClient:=RPC;
  FPoller:=TProcessCapturePoller.Create(Self);
  FPoller.OnProcessOutput:=@DoDoutput;
  FPoller.OnGetProcessStatus:=@DoGetStatus;
  FPoller.OnProcessDone:=@DoProcessDone;
  FPoller.OnStatusFail:=@DoStatusFail;
end;
```



Create Service Client Form

URL: http://localhost:8060/RPC/API

Unit name: processservice

Filename: /home/michael/source/articles/pas2js/processcontrol/client/processservice.pp

☐ Prefer NativeInt for numbers

☐ Force JSValue Result type

Preview

```
 ---------------------------------------------------------------------}

TprocesscontrolService = Class(TRPCCustomService)
Protected
  Function RPCClassName : string; override;
Public
  Function GetStatus (processID : String; aOnSuccess : TJSValueResultHandler = Nil; aOnFailure :
TRPCFailureCallBack = Nil) : NativeInt;
  Function StartProcess (a : String; b : String; aOnSuccess : TJSValueResultHandler = Nil; aOnFailure :
TRPCFailureCallBack = Nil) : NativeInt;
end;

implementation
```

☐ Add unit to project

☐ Copy source to clipBoard in addition to file

Figure 6: The service generation dialog

Note that we assign the **RPC** client to our service definition, and that we assign events to all event handlers of the poller component.

To start the process, we add an `OnClick` event handler to the `actbtnStart` action.In it, we collect the values for the A and B parameters from the respective input boxes, and use these to call `StartProcess` on our Service component.

We take care to handle the `OnSuccess` and `OnFail` handlers of this method - remember, the calls to the server are asynchronous:

```
procedure TfrmIndex.actbtnStartExecute(Sender: TObject; Event: TJSEvent);

  procedure DoStartFail(Sender: TObject; const aError: TRPCError);
  begin
    Writeln(’Failed to start process : ’,aError.Message);
  end;

  procedure DoStartOK(aResult: JSValue);
  begin
    FJobID:=String(aResult);
    FPoller.ProcessID:=FJobID;
    FPoller.Start;
  end;

var
  a,b : string;

begin
  a:=actedtA.Value;
  b:=actedtB.Value;
  Service.StartProcess(A,B,@DoStartOK,@DoStartFail);
end;
```

If the start call fails, we simply log the fact. If the start call succeeds, we record the result (a process ID) in the poller `ProcessID` property and start the poller. The onclick handler for the ’**Cancel**’ button is much simpler:
We just need to cancel the `poller`.

```
procedure TfrmIndex.actbtnCancelExecute(Sender: TObject; Event: TJSEvent);
begin
  Writeln(’Canceled wait for process.’);
  FPoller.Cancel;
end;
```

All that remains to do is to handle the 4 events of the `TProcessCapturePoller` component.
We’ll start with the simple ones, the `OnProcessOutput` and OnStatusFail events. In it, we just need to output the messages that are passed to the event handler:

```
procedure TfrmIndex.DoStatusFail(Sender: TObject; aError: String);
begin
  Writeln(’Error getting status: ’,aError);
end;

procedure TfrmIndex.DoDoutput(Sender: TObject; aOutput: String);
begin
  Writeln(aOutput);
end;
```

The `OnProcessDone` event handler is equally simple, we print the status and exit code (*if there is one*)

```
procedure TfrmIndex.DoProcessDone(Sender: TObject;
                                    aStatus: TProcessStatus;
                                    aExitCode: Integer);
Const
    Exits : Array[TProcessStatus] of string
            = (’Running’,’Exited’,’Error’);
begin
    Write(’Process ’,Exits[aStatus]);
    if aStatus=psExited then
        Writeln(’ with exit code ’,aExitCode)
    else
        Writeln();
end;
```

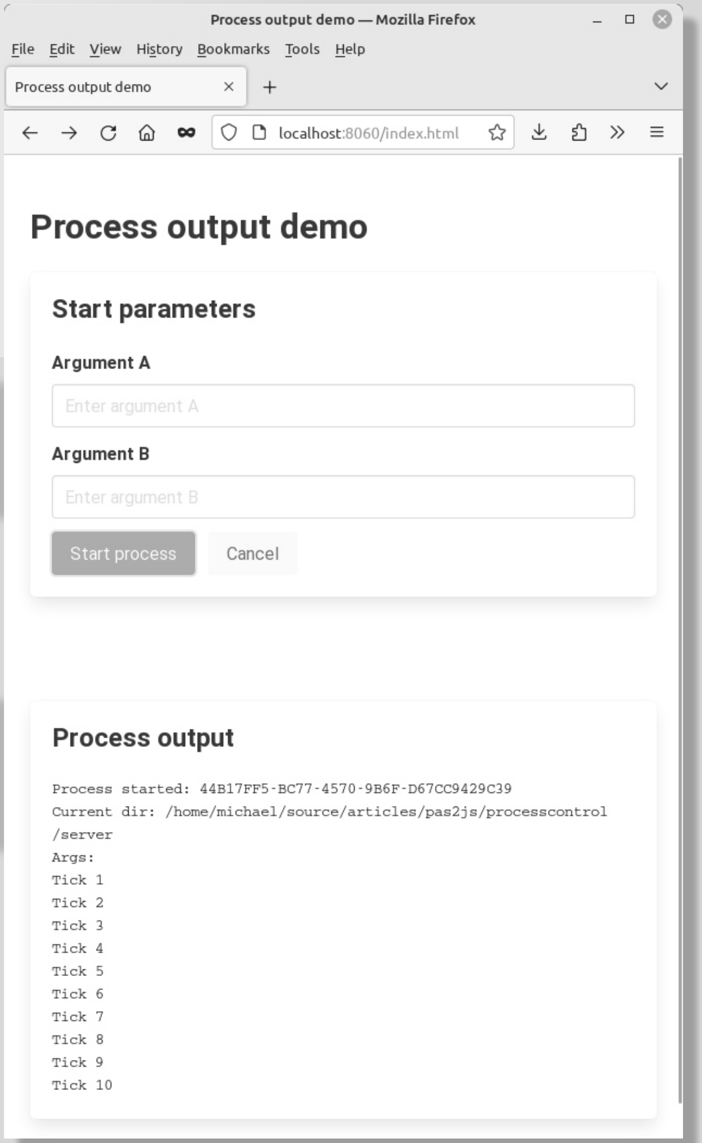
Last but not least, we must handle the `OnGetProcessStatus` event.
This simply calls the `GetStatus` procedure from our service component, and handles the result handlers: in each handler the appropriate method of the `TProcessCapturePoller` component is called with the received result:

```
procedure TfrmIndex.DoGetStatus(Sender: TObject;
                                aProcessID: String;
                                aOffset: NativeInt);

procedure DoStatusFail(Sender: TObject; const aError: TRPCError);
begin
  FPoller.ReportProgressFail(aError.Message);
end;

procedure DoStatusOK(aResult: JSValue);

const statuses : array[Boolean] of TProcessStatus
             = (psError,psRunning);

Var
  D : TJSObject absolute aResult;
  aExitCode : Integer;
  aNewOffset : NativeInt;
  aOutput    : string;
  aStatus    : TProcessStatus;

begin
  aOutput   :=String(D[’output’]);
  aExitCode :=NativeInt(D[’status’]);
  aNewOffset :=NativeInt(D[’offset’]);
  aStatus   :=Statuses[aExitCode=-1];
  FPoller.ReportProgress(aStatus,
           aOutput,aExitCode,aNewOffset)
end;

begin
  Service.GetStatus(FJobID,aOffset,
           @DoStatusOK,@DoStatusFail);
end;
```

With this, the logic of our application is ready. Remains to write the main program routine, which is very short indeed: All we need to do is create our module and call Show:

```
var
  frm : TfrmIndex;
begin
  MaxConsoleLines:=15;
  frm:=TfrmIndex.Create(Nil);
  Frm.Show;
```

Setting the `MaxConsoleLines` to `15` will make sure you can see the messages scroll over the screen as the output of the server process comes in. The result of this code is shown in figure 7 on page 15 of the article.



Process output demo

Start parameters

Argument A

Argument B

Process output

```
Process started: 44B17FF5-BC77-4570-9B6F-D67CC9429C39
Current dir: /home/michael/source/articles/pas2js/processcontrol
/server
Args:
Tick 1
Tick 2
Tick 3
Tick 4
Tick 5
Tick 6
Tick 7
Tick 8
Tick 9
Tick 10
```

Figure 7: The program in action

8 CREATING A SERVER PROCESS EXECUTION COMPONENT

Earlier in the article we mentioned that it could seem strange that there are events to report status and output when the actual call to get the status is executed in the form module: at that point you will already know the status, so why still report it to the component ?

Part of the answer is that what we have shown above is just one way to use the component. A second way is that you can also create a descendent of this component which handles the getting of the status all by itself. In that case, the events are the only way to get notifications of the status of the process. In the following we show how to make such a descendent.

The `TProcessCapturePoller` component is actually a simple descendent of the `TCustomProcessCapturePoller` component, which simply implements the method to get the status of the process using an event.

What we can do is create a descendent of the `CustomProcessCapturePoller` component which has the `TprocesscontrolService` class built-in. This component will know all by itself how to execute a process on the server. This component would look as follows:

```
TRemoteExecutor = class(TCustomProcessCapturePoller)
Protected
   procedure DoStatusCheck; override;
Public
   Procedure Execute(a,b : String);
Published
   Property RPCClient : TRPCClient Read GetClient Write SetClient;
   Property OnProcessDone;
   Property OnProcessOutput;
   Property OnStatusFail;
   Property LinebasedOutput;
   Property PollInterval;
   Property MaxFailCount;
   Property MaxCheckCount;
end;
```

We left out the constructor and destructor, which simply create and destroy the `TprocesscontrolService`.

```
constructor TRemoteExecutor.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  FService:=TprocesscontrolService.Create(Self);
end;

destructor TRemoteExecutor.Destroy;
begin
  FreeAndNil(FService);
  inherited Destroy;
end;
```

The Service field is used to get and set the `RPCClient` property:

```
function TRemoteExecutor.GetClient: TRPCClient;
begin
 Result:=FService.RPCClient;
end;

procedure TRemoteExecutor.SetClient(AValue: TRPCClient);
begin
  FService.RPCClient:=aValue;
end;
```

The `Execute` method takes the correct parameters, and in essence does what was done in the form in our original code:

```
procedure TRemoteExecutor.Execute(a, b: String);

  procedure DoStartFail(Sender: TObject; const aError: TRPCError);
  begin
    SetFailCount(MaxFailCount);
    ReportProgressFail(aError.Message);
  end;

  procedure DoStartOK(aResult: JSValue);
  begin
    ProcessID:=String(aResult);
    Start;
  end;

  begin
    Service.StartProcess(A,B,@DoStartOK,@DoStartFail);
  end;
```

Note that if the process failed to start, the fail count is set to the maximum, this will cause the `ReportProgressFail` method not to schedule a new check. The `DoStatusCheck` method contains simply the code that was present in the form in our first implementation:

```
procedure TRemoteExecutor.DoStatusCheck;

   procedure DoStatusFail(Sender: TObject; const aError: TRPCError);
   begin
     ReportProgressFail(aError.Message);
   end;

   procedure DoStatusOK(aResult: JSValue);

   const statuses : array[Boolean] of TProcessStatus
                  = (psError,psRunning);
Var
     D : TJSObject absolute aResult;
     aExitCode : Integer;
     aNewOffset : NativeInt;
     aOutput   : string;
     aStatus   : TProcessStatus;
   begin
     aOutput   :=String(D[’output’]);
     aExitCode :=NativeInt(D[’status’]);
     aNewOffset :=NativeInt(D[’offset’]);
     aStatus   :=Statuses[aExitCode=-1];
     DoReportProgress(aStatus,aOutput,aExitCode,aNewOffset)
   end;
begin
   service.GetStatus(ProcessID,OutputOffset,@DoStatusOK,@DoStatusFail);
end;
```

The form code is now much simpler. We only need to create the `TRemoteExecutor` component, and set its 3 events:

```
procedure TfrmIndex.DataModuleCreate(Sender: TObject);
begin
   FRemote:=TRemoteExecutor.Create(Self);
   FRemote.OnProcessOutput:=@DoDoutput;
   FRemote.OnProcessDone:=@DoProcessDone;
   FRemote.OnStatusFail:=@DoStatusFail;
end;
```

The event handler for the 'Start' button is now a simple one-liner:

```
procedure TfrmIndex.actbtnStartExecute(Sender: TObject; Event: TJSEvent);

begin
  FRemote.Execute(actedtA.Value,actedtB.Value);
end;
```

The event handler to get the status is no longer needed.
The functional working of the program is not different, but if you have a lot of locations in your program where you need to execute programs on the server, it makes sense to abstract away the remote execution in this manner.

❾ CONCLUSION

In this article we've shown that executing programs on a HTTP Server from a **Pas2JS** program does not need to be difficult. The component to automate the process is independent of a **RPC** mechanism, and as such can be used as-is, or it can be used as the parent for a more elaborate component which handles all communication by itself.

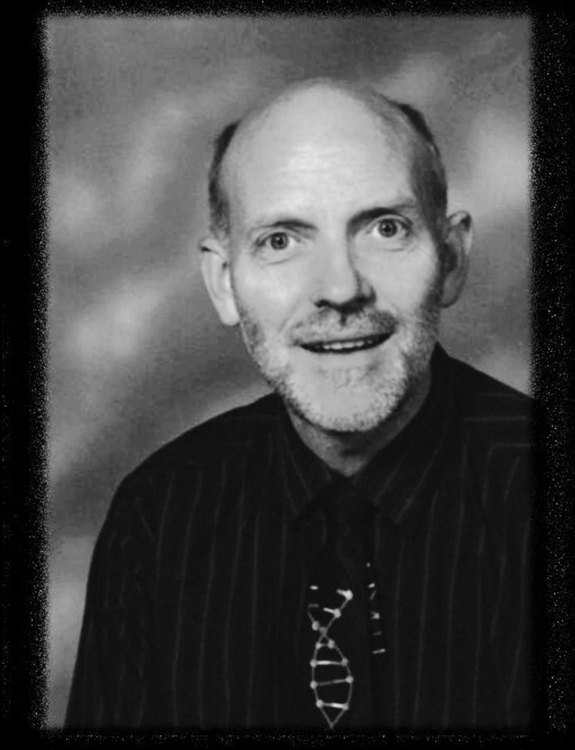
* * *


barnsten
RAD

A very great friend, collaborator, translator and proofreader as well as an important author who helped us greatly over the years - as he was always ready to help everyone - has nevertheless been extremely quickly overwhelmed by a final irrevocable stage of Cancer after a long history of illness.

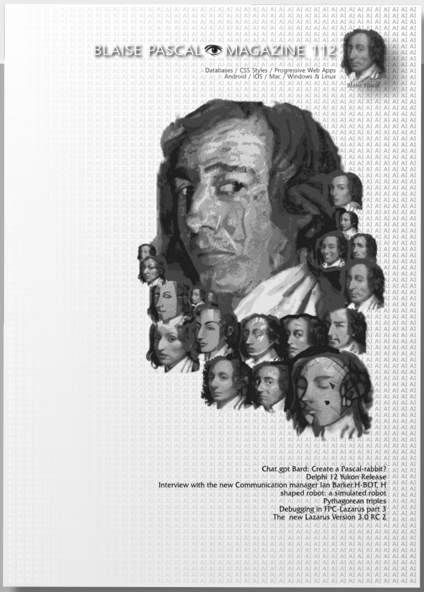
We had always hoped for a recovery as he had had before.
Sadly, he has now passed away.
We will certainly miss him extremely with his incredible power of expression in the English language and wonderful phrasing.

Howard Page Clark



ADVERTISEMENT

1

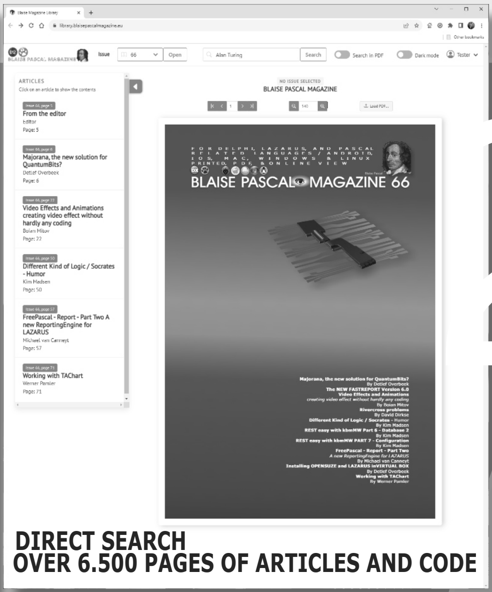


BLAISE PASCAL MAGAZINE 112

Databases / CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

Chat.gpt Bard: Create a Pascal-rabbit?
Delphi 12 Yukon Release
Interview with the new Communication manager Ian Barker.H-BOT, H
shaped robot: a simulated robot
Pythagorean triples
Debugging in FPC-Lazarus part 3
The new Lazarus Version 3.0 RC 2

2



ARTICLES
Click on an article to show the contents

Issue 66, page 5
From the editor
Editor
Page: 5

Issue 66, page 6
Majorana, the new solution for QuantumBits?
Detlef Overbeek
Page: 6

Issue 66, page 22
Video Effects and Animations creating video effect without hardly any coding
Boian Mitov
Page: 22

Issue 66, page 50
Different Kind of Logic / Socrates - Humor
Kim Madsen
Page: 50

Issue 66, page 57
FreePascal - Report - Part Two A new ReportingEngine for LAZARUS
Michael van Canneyt
Page: 57

Issue 66, page 71
Working with TAChart
Werner Pamler
Page: 71

FOR DELPHI, LAZARUS, AND PASCAL RELATED LANGUAGES / ANDROID, IOS, MAC, WINDOWS & LINUX PRINTED, PDF, & ONLINE VIEW

BLAISE PASCAL MAGAZINE 66

Majorana, the new solution for QuantumBits?
By Detlef Overbeek
The NEW FASTREPORT Version 6.0
Video Effects and Animations
creating video effect without hardly any coding
By Boian Mitov
Rivercross problems
By David Dirkse
Different Kind of Logic / Socrates - Humor
By Kim Madsen
REST easy with kbmMW Part 6 - Database 2
By Kim Madsen
REST easy with kbmMW PART 7 - Configuration
By Kim Madsen
FreePascal - Report - Part Two
A new ReportingEngine for LAZARUS
By Michael van Canneyt
Installing OPENSUZE and LAZARUS inVIRTUAL BOX
By Detlef Overbeek
Working with TAChart
By Werner Pamler

DIRECT SEARCH
OVER 6.500 PAGES OF ARTICLES AND CODE

3

LAZARUS HANDBOOK
POCKET
Edition
+shipment

4

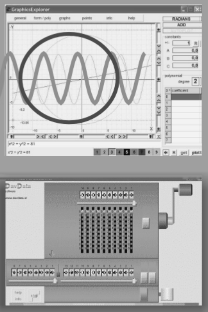
LAZARUS
HANDBOOK
PDF

5

LEARN TO PROGRAM USING LAZARUS
HOWARD PAGE-CLARK

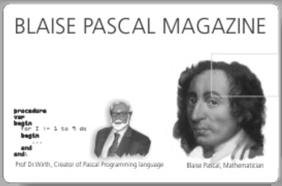
6

DAVID DIRKSE
including 50 example projects



BLAISE PASCAL MAGAZINE

COMPUTER (GRAPHICS)
MATH & GAMES IN
PASCAL



BLAISE PASCAL MAGAZINE

```
procedure
var
begin
  for I := 1 to 9 do
  begin
    ...
  end
end;
```

Prof Dr.Wirth, Creator of Pascal Programming language
Blaise Pascal, Mathematician

BLAISE PASCAL MAGAZINE

Prof Dr.Wirth, Creator of Pascal Programming language
Blaise Pascal, Mathematician

Editor in Chief: Detlef Overbeek
Edelstenenbaan 21 3402 XA
IJsselstein Netherlands

Prof Dr.Wirth, Creator of Pascal Programming language

editor@blaisepascalmagazine.eu
https://www.blaisepascalmagazine.eu

SUMMER SUPER PACK 6 ITEMS 2023

1. One year Subscription
2. The newest LIB Stick
  - All issues 1-111
  - On Credit Card
3. Lazarus Handbook Pocket
4. LH PDF including Code
5. Book Learn To Program
  - using Lazarus PDF including 19 lessons and projects
6. Book Computer Graphics Math & Games
  - PDF including ±50 projects

PRICE € 120

NORMAL PRICE € 275


Blaise Pascal

Donate for Ukraine and get a free license at:
https://components4developers.blog/2022/02/26/donate-to-ukraine-humanitarian-aid/

kbmMW Professional and Enterprise Edition v. 5.22.10
kbmMemTable v. 7.98.00
Standard and Professional Edition

5.22.00 is a release with containing new stuff, refinements and bugfixes.,**OpenSSL v3 support,** WebSocket support, further improvements to SmartBind, new high performance hashing algorithms, improved RemoteDesktop sample and much more.

This release requires the use of **kbmMemTable** v. 7.97.00 or newer.

- RAD Alexandria supported
- Win32, Win64, Linux64, Android, IOS 32, IOS 64 and OS X client and server support
- Native high performance 100% developer defined application server
- Full support for centralised and distributed load balancing and fail-over
- Advanced ORM/OPF support including support of existing databases
- Advanced logging support
- Advanced configuration framework
- Advanced scheduling support for easy access to multi thread programming
- Advanced smart service and clients for very easy publication of functionality
- High quality random functions.
- High quality pronounceable password generators.
- High performance LZ4 and J peg compression
- Complete object notation framework including full support for YAML, BSON, Messagepack, J SON and XML
- Advanced object and value marshalling to and from YAML, BSON, Messagepack, JSON and XML
- High performance native TCP transport support
- High performance HTTPSys transport for Windows.
- CORS support in REST/HTML services.
- Native PHP, Java, OCX, ANSI C, C#, Apache Flex client support!

kbmMemTable is the fastest and most feature rich in memory table for Embarcadero products.

- Easily supports large datasets with millions of records
- Easy data streaming support
- Optional to use native SQL engine
- Supports nested transactions and undo
- Native and fast build in M/D, aggregation/grouping range selection features
- Advanced indexing features for extreme performance

- New: full Web-socket support.
  The next release of kbmMW Enterprise Edition will include several new things and improvements.
  One of them is full Web-socket support.
- New I18N context sensitive internationalisation framework to make your applications multilingual.
- New ORM LINQ support for Delete and Update.
  Comments support in YAML.
- New StreamSec TLS v4 support (by StreamSec)
  Many other feature improvements and fixes.

Please visit http://www.components4developers.com
for more information about kbmMW

- High speed, unified database access (35+ supported database APIs) with connection pooling, metadata and data caching on all tiers
- Multi head access to the application server, via REST/AJAX, native binary, Publish/Subscribe, SOAP, XML, RTMP from web browsers, embedded devices, linked application servers, PCs, mobile devices, Java systems and many more clients
- Complete support for hosting FastCGI based applications (PHP/Ruby/Perl/Python typically)
- Native complete AMQP 0.91 support (Advanced Message Queuing Protocol)
- Complete end 2 end secure brandable Remote Desktop with near realtime HD video, 8 monitor support, texture detection, compression and clipboard sharing.
- Bundling kbmMemTable Professional which is the fastest and most feature rich in memory table for Embarcadero products.

COMPONENTS
DEVELOPERS 4

D11

ADVERTISEMENT